Robusta: An approach to building dynamic applications

Walter Rudametkin

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Walter Andrew RUDAMETKIN IVEY

Thèse dirigée par **Jacky ESTUBLIER**

préparée au sein du **Laboratoire d'Informatique de Grenoble** dans **L'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Robusta : Une approche pour la construction d'applications dynamiques

Thèse soutenue publiquement le **21/02/2013**,

devant le jury composé de :

M. Noel DE PALMA
Professeur à l'Université Joseph Fourier, Président

M. Benoit BAUDRY
Chargé de Recherche, INRIA Rennes, Rapporteur

M. Luciano BARESI
Professeur au Politecnico di Milano, Rapporteur

M. Eric GRESSIER-SOUDAN
Professeur au CNAM, Examinateur

M. François EXERTIER
Responsable de l'équipe JOnAS à Bull S.A.S., Examinateur

M. Jacky ESTUBLIER
Directeur de Recherche au CNRS, Directeur de thèse

Dedication

I dedicate this dissertation to my loving family and my beautiful wife.

Acknowledgements

Thank you everybody!

You know who you are.

You know how you helped.

There are so many people.

Thank you for everything.

I couldn't have done it without you.

Each and every one of you.

I sincerely thank you.

Robusta: An approach to building dynamic applications

Abstract

Current areas of research, such as ubiquitous and cloud computing, consider execution environments to be in a constant state of change. Dynamic applications—where components can be added, removed and substituted during execution—allow software to adapt and adjust to changing environments, and to accommodate evolving features. Unfortunately, dynamic applications raise design and development issues that have yet to be fully addressed.

In this dissertation we show that dynamism is a crosscutting concern that breaks many of the assumptions that developers are otherwise allowed to make in classic applications. Dynamism deeply impacts software design and development. If not handled correctly, dynamism can silently corrupt the application. Furthermore, writing dynamic applications is complex and error-prone, and given the level of complexity and the impact dynamism has on the development process, software cannot become dynamic without (extensive) modification and dynamism cannot be entirely transparent (although much of it may often be externalized or automated).

This work focuses on giving the software architect control over the level, the nature and the granularity of dynamism that is required in dynamic applications. This allows architects and developers to choose where the efforts of programming dynamic components are best spent, avoiding the cost and complexity of making all components dynamic. The idea is to allow architects to determine the balance between the efforts spent and the level of dynamism required for the application's needs.

At design-time we perform an impact analysis using the architect's requirements for dynamism. This serves to identify components that can be corrupted by dynamism and to—at the architect's disposition—render selected components resilient to dynamism. The application becomes a well-defined mix of dynamic areas, where components are expected to change at runtime, and static areas that are protected from dynamism and where programming is simpler and less restrictive.

At runtime, our framework ensures the application remains consistent—even after unexpected dynamic events—by computing and removing potentially corrupt components. The framework attempts to recover quickly from dynamism and to minimize the impact of dynamism on the application.

Our work builds on recent Software Engineering and Middleware technologies—namely, OSGi, iPOJO and APAM—that provide basic mechanisms to handle dynamism, such as dependency injection, late-binding, service availability notifications, deployment, lifecycle and dependency management. Our approach, implemented in the Robusta prototype, extends and complements these technologies by providing design and development-time support, and enforcing application execution consistency in the face of dynamism.

Table of Contents

List of Figures

List of Tables

Part I:

Introduction

Chapter 1

Introduction

"To study in depth an aspect of one's subject matter in isolation […] all the time knowing that [it is] only one of the aspects […] is what I sometimes have called 'the separation of concerns'."
—Dijkstra, "On the role of scientific thought", 1974.

1.1 Motivations and Overview

Dynamic applications—where components can be added, removed and substituted during execution—are becoming an ever more important part of software engineering. Adapting software at runtime has enormous potential for increasing an application's capacity to adjust to changing execution environments, to become more resilient[1], and to continuously accommodate evolving features. Current areas of research, such as ubiquitous and cloud computing, are pushing the need for dynamic applications. Unfortunately, dynamic applications raise design and development issues that have yet to be fully addressed. Designing and developing dynamic applications is still mired with pitfalls that complicate the adoption of dynamism in real world systems. It is particularly difficult to decouple components sufficiently to ensure they behave properly after dynamic changes.

Current solutions to handling dynamism are split between two very different programming models: distributed solutions (*e.g.*, cloud computing, multi-process programs) where consistency and robustness guarantees for dynamism are obtained at the cost of programming complexity and often a lack of runtime efficiency; and centralized solutions, which allow for a very flexible programming model that can exploit shared memory and run very efficiently, but cannot guarantee sufficient levels of decoupling to ensure the application is robust and remains consistent in the face of dynamism.

Our approach focuses on decoupling components and ensuring proper dynamic behavior in centralized dynamic applications. More specifically, our work targets multi-threaded, synchronous, centralized[2], dynamic, component-based applications. We propose a programming model that is less restrictive than that of distributed programming, yet retains guarantees such as

[1] Resilience is the capacity of the application to continue to function correctly and provide an acceptable level of

[2] Centralized applications execute in a single address space, commonly known as a process in most operating systems. They may be multi-threaded, where threads share data and references (*e.g.*, global variables) allowing them to communicate effectively and efficiently.

the application's consistency. Given its popularity in developing complex systems, our approach focuses on components that are programmed using the Object-Oriented paradigm.

Component-based programming is a necessary step towards decreasing coupling among software entities, but it is not sufficient by itself to ensure dynamism. We use the term *static-decoupling* to describe the level of decoupling afforded by current component frameworks, which is insufficient for dynamism. Indeed, dynamism is a broad cross-cutting concern that affects architectural decisions, components' implementations (at the source code and design levels), packaging components into modules, and deployment and runtime management. Dynamism is neither transparent nor orthogonal; quite the opposite, it is very invasive. We propose *dynamic-decoupling* to achieve a level of decoupling sufficient to allow for dynamism and still retain consistency guarantees.

Notwithstanding the complexities in programming dynamic components, dynamism is not necessary for every component. Moreover, the same component used in different applications may require different levels of dynamism or resilience to dynamic change. It is not difficult to conceive that, in a dynamic application, some components require being added, removed or substituted, while many others simply do not (*e.g.*, the application's core components). Dynamism can and should be selectively pursued and highly targeted. Simply put, there is a clear tradeoff between development effort and dynamism—development efforts should focus on dynamic concerns where it benefits the application the most.

In large and complex systems, we believe the software architect is best positioned to decide the application's dynamic requirements. This allows decisions regarding the tradeoff between effort and dynamism to be made explicit in the architecture at the architect's discretion. Software architecture approaches, based on component approaches themselves, provide a level of abstraction that is useful for structuring and reasoning about software. Software architecture moves focus from *programming-in-the-small* to *programming-in-the-large*, directing developers' and architects' attention from low-level implementation details to high-level integration and design concerns. Software architectures are ideal for reasoning about dynamism.

Executing dynamic applications becomes notably punctilious given this flexible programming and design model for dynamism. The runtime can no longer suppose everything is simply the same, that is, that components are programmed either purely statically or completely dynamically. There is a large gray area of components in between that can react differently to the type of dynamic change at hand. The levels of resilience and the dynamic behavior components will show at runtime are different because the components have been designed and programmed with these varying dynamic restrictions and requirements. The runtime must interpret design-time metadata with the purpose of properly handling the application's execution. Reflexive component models assist us in reifying design-time concepts at runtime, allowing for a better integration of the commonly distinct processes that begin to merge. Reifying and sharing concepts between runtime and design-time aides in the analysis of design-time dynamism decisions and their effects on running applications.

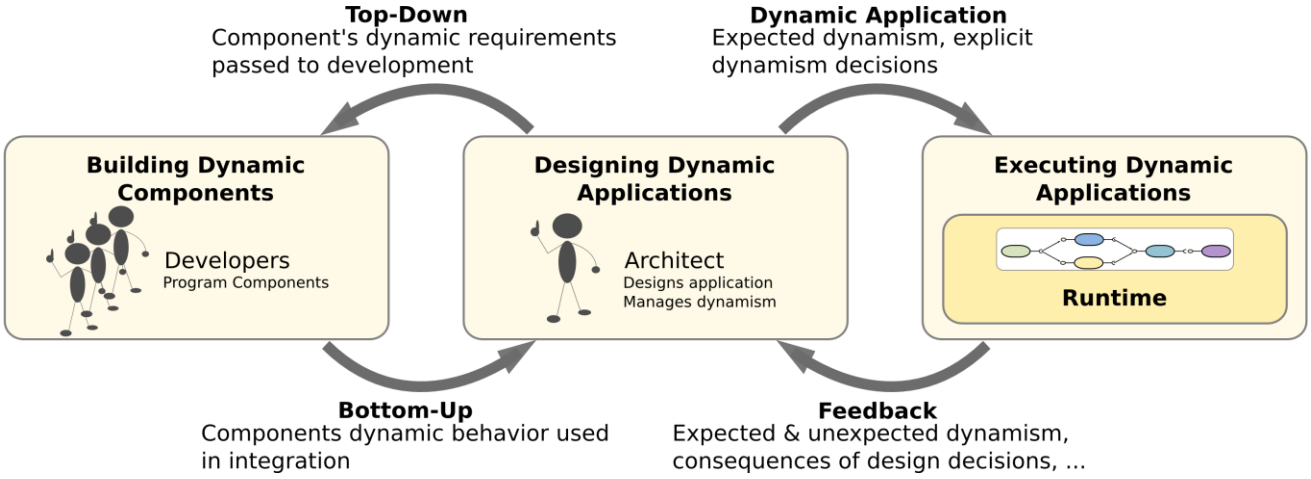

Figure 1: The process of designing, building and executing dynamic applications.

The role architects and developers play in the design and construction of dynamic applications are cyclic and interactive. Figure 1 shows the relationship between developers, architects and the application's runtime. Architects are central to our approach and are empowered with the design and control of the application's dynamism. In a top-down fashion, from specification to implementation, architects dictate the dynamic requirements of components and the levels of resilience of their dependencies. In essence, they specify the dynamic behavior of the components. Developers implement components following the architect's specifications. However, an architect is also tasked with integrating existing components into dynamic applications, forcing the architect to deal with a component's existing dynamic behavior. Indeed, the process between developing the components and designing the application's architecture is cyclic and based on gradual refinement; both a process of integration and one of specification are necessary. Once a dynamic application has been designed and built, it is executed. Indeed, following our approach of selective dynamism to fit the application's needs, the architect builds the application with expected points of dynamism which are made explicit in the architecture. The runtime must execute the application and, when dynamic events occur, make the necessary changes to the application to ensure it continues running while still remaining consistent. This process is also cyclic. The runtime ensures consistency and eliminates potentially corrupt components, which is possible thanks to having knowledge of the dynamic behavior and resilience the components have been implemented with. This information serves architects in the design and improvement of the dynamic application. Yet, because not all components are resilient to dynamism—the application has been built selectively implementing dynamism—some components may be corrupted and need to be removed for the application to continue executing properly. The consequences of design decisions and the results of dynamic change are used to improve and refine the architecture.

Silent corruption—a risk when programming centralized dynamic applications—can lead to memory leaks or unexpected and undesirable behavior. Dynamic decoupling and resilience ensures that corruption does not occur by clearly separating components from one another, allowing them to be independently added, removed or substituted. However, the effort of decoupling is selective and optional; it is a dynamic requirement that is specified by the architect and used where dynamism is expected and deemed useful. In spite of the architect carefully enabling dynamism, the origins of dynamism are impossible to completely predict and control.

Unexpected dynamism (*e.g.*, a forced update to a component, a failure) can and does occur, putting at risk of corruption components that are not resilient to change because they were not expected to change, leaving a choice to be made about what to do with such components. This involves a second tradeoff, the risk of continuing to use a potentially corrupted component versus the downtime involved in replacing it—availability versus consistency. We believe that consistency always trumps availability and should be enforced. It is preferable to err on the side of safety. The risk of corruption caused by dynamism is too great and, in our minds, an important deterrent to building dynamic applications that we attempt to overcome. The runtime, when a dynamic change occurs, calculates the propagation of corruption through the components and across the architecture and removes all potentially corrupt components. Design-time metadata and the components' dynamic characteristics allow such calculations to be reasonably precise. Furthermore, the runtime attempts to minimize the impact of dynamism on the running application by gracefully passivating components that are to be removed and branching the architecture at safe-points to new components.

Our work throughout this dissertation is focused on the following aspects:

- From a dynamic application's point of view:
  - Determine the inhibitors to effectively using dynamism in current Component Models, both in theory and in practice
  - Determine the requirements to design and program dynamic components
  - Understand the roles architects and developers play when constructing dynamic software
- From an architect's point of view:
  - Promote dynamism into the design and management of software architectures, where it can be handled as an architectural-concern instead of in an ad-hoc manner in each component
  - Allow selectively enabling dynamism where it benefits the application most in order to minimize complexity and wasted effort, and protect sensitive zones of the application from the instability generated from dynamism
  - Provide analyses to verify the architect's assumptions about an application's dynamic behavior
- From a developer's point of view:
  - Provide an approach and guidelines for the construction of dynamic components that ensure they are sufficiently decoupled to ensure proper dynamic behavior in spite of dynamism
  - Provide tools for assisting and verifying dynamic components are properly programmed
- From a runtime's point of view:

- Produce a framework that manages the application's dynamic behavior during execution, distinguishing between dynamic components, their levels of resilience to change and statically programmed components
- Ensure the application remains consistent despite the effects of expected and, more importantly, unexpected dynamism
- Minimize the impact of dynamism at runtime

Finally, our work builds on recent Software Engineering and Middleware technologies like iPOJO[3] and APAM[4] that provide basic mechanisms to handle dynamism, such as dependency injection, late-binding, service availability notifications, deployment, lifecycle and dependency management. Furthermore, as do both iPOJO and APAM, we rely on the OSGi[5] framework as a dynamic module system that enables deployment. Our approach, implemented in the Robusta prototype, extends and complements these technologies by providing design and development-time support, and enforcing application execution consistency in the face of dynamism.

1.2 Dissertation Structure

The remainder of this document is divided into three parts, namely, the state of the art, our approach to building dynamic applications, and our conclusions and perspectives. This section presents an overview of each chapter of the document.

Chapter 2 presents the general concepts and background information useful for understanding this work. Namely, we introduce the concepts of Software Architecture, CBSE, Modules, Service Oriented Computing and Service-Oriented Components.

Chapter 3 introduces Software Evolution and compares it to Software Maintenance. Evolution is part of the development process and plays a special role in regards to software architecture.

Chapter 4 moves onto Dynamic Software Evolution, which involves changing and adapting software at runtime. We go over similar approaches to architecture-based software evolution and we present the main research issues.

Chapter 5 presents Robusta, our approach to building dynamic applications. We present the main concepts we use and how our approach functions at a high-level of abstraction.

Chapter 6 explains what Dynamic Decoupling is and how it works. We describe how to decouple component implementations such that they can be added, removed, or substituted individually. We also describe the restrictions on decoupling component instances such that they continue to function properly should their dependencies be changed. Our approach focuses on the Service Contract concept and describes the insufficiencies caused by reducing the contract to a simple Service Interface.

[3] http://felix.apache.org/site/apache-felix-ipojo.html
[4] http://wikiadele.imag.fr/index.php/APAM
[5] http://www.osgi.org/

Chapter 7 details how we protect components from failure and dynamism by means of isolation barriers and recovery mechanisms. We also detail the mechanisms necessary at runtime to ensure consistency. These mechanisms and calculations are shared at design-time in order for architects to understand the expected dynamic behavior their applications will exhibit.

Chapter 8 provides an overview of our approach, from design to runtime and back. We also describe the types of analysis that can be performed at the architectural level, as well as at the component implementation level, to assist architects and developers respectively in their quest to build dynamic applications.

Chapter 9 describes the implementation and validation of our approach, the Robusta framework. Robusta relies directly on the APAM framework for designing, executing, deploying and running dynamic applications, and indirectly relies on the iPOJO component model and OSGi dynamic module platform.

Chapter 10 presents our conclusion and the perspectives of this work.

Part II:

State of the Art

Chapter 2

Background

"Those who forget the past are doomed to repeat it."
—Adage

2.1 Introduction

In this chapter we present the main background concepts regarding this dissertation's area of interest, namely, that of dynamic applications and dynamic components.

Modern software and their increasing size and complexity have continuously pushed the boundaries of what was thought as feasible [Northrop *et al.* 2006]. Software architectures have come to be recognized as one of the most promising solutions for mitigating complexity [D. Perry and Wolf 1992][Richard N. Taylor and Van der Hoek 2007]. The design and specification of the overall structure of a system becomes a critical issue and a decisive factor in its success or failure. Architecture design can impact performance, reliability, scalability, interoperability, maintainability and portability [Garlan 2002].

Software architecture describes structure, the key elements of a software system, its organization and interaction. Seen as a discipline, it is a key issue in the development of large systems. Software architecture focuses on *programming-in-the-large* rather than *programming-in-the-small* [DeRemer and Kron 1975], and as such takes a step back from low-level details related to algorithms and data (*e.g.*, variables, types, constants) and focuses on architecture (*e.g.*, components, modules, interfaces, dependency relations), variation (*e.g.*, variants, compatibility) and evolution [Favre 1997]. In general, software architecture helps us to understand the system while hiding low-level details.

This dissertation focuses on managing dynamism in dynamic applications and in giving architect's control over dynamism in the software architectures. Background concepts related to our goal, such as software architecture, components, modules, and others are explained in this chapter.

2.2 Software Architecture

Researchers began to focus on software design in the '70s [Wasserman 1990; Bergland 1981]. This was because during the 60's initial problems unique to large scale systems were being discovered and gaining recognition [Brooks 1995]. The importance of these issues rapidly prompted researchers to differentiate between the processes of implementation and design, both requiring their proper techniques and tools, which notably lead towards Computer Aided Software Engineering technology (CASE) [Premkumar and Potter 1995]. In the 80's software design seemed to fade away while software engineering research leaned more towards integrating designs, which itself lead to a blur between implementation and design since languages began to integrate previous notations and techniques for large systems. In the early 90's, Perry and Wolf [D. Perry and Wolf 1992] introduce a deep contrast between the ambitions of software design and those of software architecture. Software architecture is supported by notions of codification and abstraction, by formal training, by standards and by style. They conclude their work with a simile in an attempt to give insight into why software systems are so difficult to evolve and by promoting architecture as a solution and an interesting subject for future research:

> *"Perhaps the reason for such slow progress in the development and evolution of software systems is that we have trained carpenters and contractors, but no architects".*
>
> *[D. Perry and Wolf 1992]*

These initial works, driven by the intuition that there was an important aspect of software engineering that had yet to be properly addressed and on which the future of ever more complex systems would need to rely on, gave way to a flurry of interest in the domain. Nowadays, we can look back and see with a much clearer eye the common ground regarding the concepts, techniques and methodologies that have been found (*e.g.*, component, connector, configuration, binding).

Some of the works of interest that further contributed to the foundations of software architecture as a discipline include the first book written on the subject, by Shaw and Garlan [Mary Shaw and Garlan 1996], which provides an overview of industrial and research projects along with a large collection of relevant definitions. Other books that contributed to the discipline began to specialize on certain aspects, as for example, on software architecture patterns [Buschmann *et al.* 1996], architecture modeling [Hofmeister *et al.* 1999], and architecture evaluation [Paul Clements *et al.* 2002]. Further research leading to the consolidation of software architecture as a research discipline in software engineering include [M. Shaw and P. Clements 2006], [Richard N. Taylor and Van der Hoek 2007], and [R. N. Taylor *et al.* 2009]. Thus, architecture has become centric to the development phase and is moving into the runtime phase of software engineering, especially as the line between development and runtime blurs [Baresi and Ghezzi 2010]. Architecture will only grow in its importance across the entire lifecycle of software [R. N. Taylor *et al.* 2009].

It is well understood that a software system without an appropriate architectural design is more difficult to evolve and customize. The architecture of a system gives us much insight into the tradeoffs between the various properties that system attempts to ensure and the constraints that follow with them. Architecture styles have shown us some of the advantages when attentive detail is paid [Richard N. Taylor *et al.* 2009; Fielding 2000]. We take particular interest in two phenomena that contribute to the fragility of architectures, namely that of *architectural drift* and *architectural*

erosion. To better understand these concepts—both related to architecture degradation—we need to understand the difference between a system's *prescriptive architecture* and its *descriptive architecture*. A **prescriptive architecture** is the design decisions made prior to the system's construction, and as such, is the *as-conceived* or *as-intended* architecture. A **descriptive architecture** describes how the system is really built, *i.e.*, it is *as-implemented* or *as-realized* architecture. Ideally, a system's prescriptive architecture should be modified and then its descriptive architecture should follow. In practice, the descriptive architecture is commonly directly modified. **Architectural drift** occurs when the original design of the system and the as-implemented design diverge; yet, these design decisions are not included, encompassed or implied by the prescriptive architecture, thus, the prescriptive architecture's design decisions are not violated. **Architectural erosion** is the introduction of design decisions into the descriptive architecture that violate its prescriptive architecture. In general, architectural degradation hinders evolution, maintenance and comprehensibility [VanGurp and Bosch 2002]. If degradation occurs, it will be necessary, sooner or later, to *recover the architecture.* **Architectural recovery** is the process of determining a software system's architecture from its implementation-level artifacts [R. N. Taylor *et al.* 2009].

The rest of this chapter will go over the basic concepts regarding software architecture, including structure (a system's structure is described by architecture elements, *i.e.*, components and connectors, and their interactions) and description languages (which are used to describe structure).

2.2.1 Definitions for Software Architecture

Various definitions for software architecture have been proposed. We will present those that we have considered the most relevant.

Perry & Wolf in 1992 define software architecture as a 3-tuple:

Software architecture = <Elements, Form and Rationale>

> ***Elements*** are the system's building blocks. There are three different classes of elements: processing elements (transform the data elements), data elements (contain the information) and connecting elements (glue that hold the pieces together).
>
> ***Form*** consists of weighted properties and relationships. Weighting distinguishes importance. Properties are used to constrain the choice of Elements (i.e., they define constraints), while relationships constrain how the different elements may interact and how they are organized.
>
> ***Rationale*** captures the motivation for the choice of architectural style, the choice of elements, and the form. The rationale explicates the satisfaction of the system constraints. These constraints are determined by considerations ranging from basic functional aspects to various non-functional aspects such as economics, performance and reliability.
>
> *[D. Perry and Wolf 1992]*

This definition is very popular and various derivations or further explanations of it have been proposed. One of the more interesting ones is from Taylor, Medvidovic and Dashofy who explain the definition in terms of *What*, *How* and *Why?*

> ***Elements*** help to answer the What questions about the architecture: What are the elements of a system? What are their primary purpose and the services they provide?
>
> ***Form*** helps to answer the How questions about the architecture: How is the architecture organized? How are the elements composed to accomplish the system's key task? How are the elements distributed?
>
> ***Rationale*** helps to answer the Why questions about the architecture: Why are particular elements used? Why are they combined in a particular way? Why is the system distributed in a given manner?
>
> *[R. N. Taylor et al. 2009]*

Furthermore, Kruchten comments on the elegance of the formula provided by Perry and Wolf, and proceeds to define software architecture as follows:

> *Software architecture deals with the design and implementation of the high-level structure of the software. It is the result of assembling a certain number of architectural elements in some well-chosen forms to satisfy the major functionality and performance requirements of the system, as well as some other, non-functional requirements such as reliability, scalability, portability, and availability.*
>
> *Software architecture deals with abstraction, with decomposition and composition, with style and esthetics.*
>
> *[Kruchten 1995]*

Another definition that was based on Perry & Wolf was that of Shaw & Garlan:

> *Software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.*
>
> *[Mary Shaw and Garlan 1996]*

A definition that considers the needs of stakeholders as a necessary concept to complete software architecture is that of Gacek *et al.*:

> *A software system architecture comprises:*
>
> - *A collection of software and system components, connections, and constraints.*
> - *A collection of system stakeholders' need statements.*
> - *A rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need-statements.*

[Gacek et al. 1995]

They argue that software architecture has a different meaning and use for different stakeholders. Given that stakeholders' needs will vary from system to system, the software-system architecture's emphasis will also vary from system to system.

The IEEE Standard provides a definition for software architecture or system architecture in 2000. IEEE 1471 is the short name for a standard formally known as ANSI/IEEE 1471-2000, Recommended Practice for Architecture Description of Software-Intensive Systems. As a framework, IEEE 1471 defines architecture (including a metamodel in UML), presents a conceptual framework and embodies a theory and practice of architectural descriptions based on that conceptual framework.

> *Architecture. The fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.*
>
> *[Mark W. Maier et al. 2004]*

Finally, a more recent definition is provided by Taylor, Medvidovic and Dashofy in their book:

> *A software system's architecture is the set of principal design decisions about the system.*
>
> *Design decisions encompass every aspect of the system under development, including: system structure, functional behaviour, interaction, nonfunctional properties and implementation.*
>
> *Principal is a term that implies a degree of importance and topicality that grants a design decision architectural status, that is, that makes it an architectural design decision (i.e. it impacts a system's architecture).*
>
> *How one defines "principal" will depend on what the stakeholders define as the system goals.*
>
> *[R. N. Taylor et al. 2009]*

This definition places software architecture at the heart of the development process. Software architecture is the blueprint for a software system's construction and evolution. Thus, architecture does not simply describe structure, but also behavior (*e.g.*, data processing, storage, visualization), interaction (*e.g.*, synchronous or asynchronous communication, procedure-based communication, RPC), extra-functional properties (*e.g.*, quality of service) and technology (*e.g.*, Linux, Python, Java). Furthermore, Taylor *et al.*, in their recent book, have opted to provide liberal definitions for the various concepts related to software architecture, turning it into a more all-enclosing concept, which is consistent with their desire to extend its use.

Although each definition has its particular focus or uniqueness, there are two main characteristics to all of them. Software architecture refers to structure and behavior. Structure describes how the system's building blocks—which we will now call *components*— are assembled. Behavior is the visible interaction of the systems components to achieve a functional system. Together, they define the software's architecture. Furthermore, they can be formally described using an Architecture Description Language or ADL, which we will explain later.

2.3 Component-Based Software Engineering

Software Engineering is, relatively speaking, a young discipline, especially when compared to other engineering disciplines, such as civil engineering or mechanical engineering. Modern software emerged recently in human history and the fact that software is itself an intangible, abstract element does not facilitate its definition in terms of science and engineering. The novelty of this new discipline involves many abstract concepts, which are constantly evolving with our growing knowledge in the area; it is difficult to find a consensus on its definitions and terminology.

Component-based Software Engineering (CBSE) is a branch of Software Engineering in which the basic concept is that of *component*. Its goal is to bring a wide range of benefits to Software Engineering in terms of development, integration, maintenance, reusability, separation of concerns, among others. In this section, we present the key concepts of CBSE, and by inclusion, to software architecture (which we have introduced previously). Furthermore, these concepts are necessary for the proper comprehension of what an Architecture Description Language is (which we will explain in a following section), and what developers attempt to express in source code and externally (*e.g.*, in metadata). Most of these concepts are not new and have been introduced and refined over the years, and as such, we will focus on the most important definitions based on popularity, (our) preference and/or general consensus.

2.3.1 A little bit of history

Szyperski's book on components [Szyperski 1997] is one of the most cited references on component software. He is considered by many to be the precursor of the software component concept in its current form. It is not surprising that his explanations and work tend to gravitate towards the practical and concrete, being as he worked under, and was undoubtfully strongly influenced by, Niklaus Wirth at the ETH (Swiss Federal Institute of Technology), whose work includes the Pascal language [Wirth 1971], along with other important yet surprisingly less well known projects such as Modula [Wirth 1977], Oberon the language [Wirth 1988] and Oberon the operating system [Wirth 1992].

To give the reader an idea of the principles of Niklaus Wirth, in his article *A Plea for Lean Software* [Wirth 1995], Wirth expresses his fears in what we might call *software bloat* and *feature creep*. He felt that it was necessary to fight *"Fat Software"* and he explains that its tolerance comes from (i) advancing hardware speeds, (ii) customer ignorance, (iii) vendors' acceptance to continuously add more features to the peril of the system. He proposes a hard return to the essentials and to develop systems using disciplined methodologies.

Regarding Wirth, Michael Franz in a chapter called *Oberon: The Overlooked Jewel* [Böszörményi *et al.* 2000] wrote:

> *True to Wirth's maxim that software designers should be forced to use the products of their labor themselves, each of the Ph.D. students who had ported Oberon onto a new platform subsequently used Oberon on that particular platform as his main (and often sole) work environment. Only recently is this sentiment making a comeback in the software developer community, under the somewhat flippant moniker of "eating one's own dog-food". Hence, after having created the code-*

generating loader for my machine-independent code-transportation format, I actually wrote my doctoral dissertation using a version of Oberon's document processing software that was generated afresh on-the-fly by dynamic compilation each time that I started it.

Franz also went on to write:

Moreover, few people know that Ceres and Oberon not only provided the backbone of all research at the Institute for Computer System, but also for most of the education. ETH was probably the only university in the Western world that in the 1990's conducted the majority of its undergraduate education in Computer Science using workstation computers (of the Ceres family) built in-house, running an operating system (Oberon) developed in-house, and teaching a programming language (Oberon) created by one of the resident professors. And the resulting education was arguably better than anywhere else, because instead of merely explaining to students how to use the available educational computer systems, at ETH the educational system's architects were at hand to explain the motivation behind individual design decisions.

Although commonly known as *Wirth's law,* Niklaus Wirth attributes the following computing adage to Martin Reiser:

"Software is getting slower more rapidly than hardware becomes faster."

Martin Reiser later wrote in his book on the Oberon System [Reiser 1991]:

"The hope is that the progress in hardware will cure all software ills. However, a critical observer may observe that software manages to outgrow hardware in size and sluggishness."

As we see, Wirth has had an enormous influence in computer science, which has been expressed by his pupils, including a chapter by Szyperski, in the book *The school of Niklaus Wirth: the Art of Simplicity* [Böszörményi *et al.* 2000]. Szyperski compares modules and components. Modules, which are a common concept in the writings of Wirth, he argues, are not components, and components are not modules. Yet at the time there were clearly—and arguably still are—overlaps in their concepts, which he attempts to—in our opinion satisfactorily—clarify. It is interesting to note that Wirth was adding the notion of module to procedural languages (*i.e.*, Pascal) while Szyperski was working on components in the Object Oriented Paradigm [Pfister and Szyperski 1998], which led to, in our opinion, much of the confusion between the various concepts and the proposed solutions. We will clarify each definition later on in this chapter and compare the two in section 2.3.10 (Modules vs. Components), but we will avoid going further down history lane.

2.3.2 Component

As we mentioned before, Szyperski is one of the leading authors on component software. His second edition book [Szyperski *et al.* 2002] regroups fourteen different definitions for component. We will start with an early definition:

…a component is a "static abstraction with plugs"

[Nierstrasz and Dami 1995]

The authors refer to a software entity that is long lived, opaque or encapsulated, whose shell does not change, making it reusable. They call them "static", because they are inherently static and have a persistent existence independent of their context, just like many software entities, such as procedures, functions and classes. In contrast, classes may have object instances that are dynamic elements. Furthermore, it should be insertable, or plug into, the system. This implies that communication and component interaction must be specified. It is a somewhat primitive definition, yet very flexible, since we could apply the definition quite broadly.

Szyperski provides a stricter definition, which also provides more guarantees and specializes what a component is:

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*
>
> *[Szyperski et al. 2002]*

Therefore, Szyperski sees components as composition units with clear interfaces and explicit dependencies. To him, components are fundamental building blocks of software systems that will be used by third-parties, which is key to his vision on how future software will be constructed by integrating third-party, independently developed, components, which is what we commonly see nowadays thanks to the open-source and free software movements large production of reusable, *off-the-shelf* components. A definition from Meyer which was published after the latest edition of Szyperski's book is:

> *A component is a software element (modular unit) satisfying the following three conditions:*
>
> 1. *It can be used by other software elements, its "clients".*
> 2. *It possesses an official usage description, which is sufficient for a client author to use it.*
> 3. *It is not tied to any fixed set of clients.*
>
> *[Meyer 2003]*

The definition is interesting in that it clearly decouples clients from provider components, and that it requires an official usage description, which is a broader definition than Szyperski's "contractually specified interfaces".

A newer definition, as of their book in 2009, has been given by Taylor *et al.*:

> *A software component is an architectural entity that (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context.*
>
> *[R. N. Taylor et al. 2009]*

This definition is similar to Szyperski's, albeit, it does not mention individual deployment. In any case, a component is an architectural element used to structure a system, in which one can find either the system's data or its functionality (or both). Communication is restricted to well defined

interfaces, thus to clients, they appear as black-boxes. Furthermore, their dependencies need to be explicit, which is useful in analyzing a system for completeness.

2.3.3 Connector

Mary Shaw was the first to introduce explicit connectors in order to separate what she saw as different concerns, namely those of computation (components) and coordination (connectors):

> *Connectors are the locus of relations among components. They mediate interactions but are not—"things" to be hooked up (they are, rather, the hookers-up). Each connector has a protocol specification that defines its properties. These properties include rules about the types of interfaces it is able to mediate for, assurances about properties of the interaction, rules about the order in which things happen, and commitments about the interaction such as ordering, performance, etc.*
>
> *[Mary Shaw 1993]*

Shaw presents the connector as a software entity that captures the nature of component interactions. She shows us that although the system's main functional blocks are components, the properties of the system strongly depend on the character of their interactions. She proposes promoting connectors to become first-class citizens in software architecture and in their description languages.

Allen & Garlan [R. Allen and Garlan 1997] formalize the semantics of connectors, where connectors are specified as a protocol where each participant's role in the interaction is specified. Further work from [Lau and Elizondo 2005] show that connectors are a mechanism for transferring not only data but also control around a system. When a component interacts with another, by, for example, method invocation, it passes data (in parameters) and the execution control. Lau proposes to decouple communication from control using exogenous connectors.

Other studies have provided taxonomies of connectors [Mehta *et al.* 2000], while some have argued for or against the need to provide connectors as first-class citizens. The argument establishes whether or not the functionality in connectors can be, when necessary, encapsulated into other "communication" components. Thus, if special conditions for communication are necessary, such as communication cardinality (*e.g.*, fan out), then this can be included into a distinct component instead of a new entity called a connector, which simplifies the model. Bálek and Plášil argue that even if technically sane to avoid connectors, by including them into the Software Architecture (and its ADL), deployment mechanisms can generate them on-the-fly or choose them among a collection, making the system more flexible.

Connectors can provide much more functionality than just redirecting invocations between components; they can encapsulate extra-functional services, such as quality of service constraints, persistence, logging, transactions, and many others. It is likely that the advantages of adding connectors or avoiding them relates to the problem being solved. As we see in distributed communication platforms such as CORBA, .NET or RMI, such functionality are services provided by the platforms that add value and ease development, but they are not connectors. Other examples of component models that do not provide explicit connectors are Fractal [Bruneton *et al.* 2006] and iPOJO [Escoffier *et al.* 2007].

The concept of connector tends to shine in problems regarding dataflow, mediation and application integration, where complex communication patterns [Hohpe and Woolf 2004] between heterogeneous distributed components are necessary, as in the Cilia component model [Garcia *et al.* 2010].

Taylor *et al.* in chapter 5 of their book [R. N. Taylor *et al.* 2009] describe four classes of services a connecter provides:

- **Communication**: transmission of data among components
- **Coordination**: transfer of control among components
- **Conversion**: transform the interaction required by one component to that provided by another
- **Facilitation**: services that mediate and streamline component interaction. For instance, load balancing, scheduling services or concurrency control.

In any case, be there formal connectors or communication components, we do not believe this changes the essence of Software Architecture. We add an interesting reflection by Szyperski regarding pragmatic ingenuity in software developers and informally introduce the notion of adapter:

> *It is obvious that components need to be connected to be useful. It is also obvious that such connections follow standards to make it at all likely that any two components have compatible 'connectors'.*
>
> *Connection standards solve an important problem. [...] However if everything works except the wiring, then people usually find a way around this problem and call it an adapter.*
>
> *[Szyperski et al. 2002]*

In this case he talks about adapters, and, of course, standardized interfaces between components leads us to simpler integration and increased possibilities for component substitutability, breaking the system free of lock-in and making it more flexible and evolvable. Yet, we could also say that in a component world, if the right connector is not found, practicality would instead lead us to create a component that does the same job. From the perspective of our work, connectors are of little importance for the time being; we see their use more applicable to semantic analysis of component communication paths or maybe for improved causality analysis, like the work done by Aguilera *et al.* [Aguilera *et al.* 2003] in analyzing causally-related communication paths between nodes in a distributed system.

2.3.4 Composite component

Methods to describe architectures with different granularities are commonly desired. The possibility of encapsulating parts of the architecture into other components (components inside of components) helps provide a uniform view of applications at different abstraction levels. Such that, a composite component is a component with sub-components [Andrade *et al.* 2003]. In some component models the basic building block is called a *primitive-component*, and composite

components are built of primitive and/or other composite components [Bruneton *et al.* 2006]. Generally speaking, from an external perspective, a composite component is not distinguishable from a primitive component; composites add encapsulation and structure to the architecture.

Some component models that include composite components are Darwin [J. Magee *et al.* 1995], ArchWare [Oquendo *et al.* 2004], ACME [Garlan *et al.* 2000], Fractal [Bruneton *et al.* 2006] and iPOJO [Escoffier *et al.* 2007].

2.3.5 Configuration

To accomplish the system's objectives, components have to be composed (or organized) in a specific way. This represents the system's configuration, which is also referred to as its topology[6].

Taylor *et al.* provide the following definition for configuration:

> *An architectural configuration is a set of specific associations between the components and connectors of a software system's architecture.*
>
> *[R. N. Taylor et al. 2009]*

That is, a configuration is a specific structure for a concrete system. In many formalizations, the system's configuration is represented as a directed graph, wherein nodes represent components and edges represent their associations (the direction indicates who invokes who) [Hirsch *et al.* 1998]. This facilitates operations, such as calculating reconfigurations, since graph theory, and the extensive knowledge on graph transformations, can be used to solve architecture problems.

2.3.6 Ports

Ports are the communication channels that components use for interacting with each other. A component can generally have many ports, and as such, can receive information or data from different places. Ports seem to be most useful in component models that use asynchronous communication schemes. The idea of distinguishing one port from another in these cases can be important. In one such case, ports provide a useful abstraction for distinguishing data transit points because, in many asynchronous models, one cannot distinguish the data before sending it to a component because the data is generically typed (*e.g.*, it is a *message* or *standard data format*). It is also useful when the applications structure is particularly static, that is, the bindings between components have been set up for a particular reason and should not change freely.

For synchronous communication component models, the use of ports is not common. Their need arguably disappears since bindings between components use *interfaces* and each interaction is carried out using an *execution thread*, which is sufficient to distinguish the communication channels and the current interaction or interactions.

[6] We prefer and will utilize the term configuration throughout the rest of this document.

2.3.7 Bindings

Binding is the process that establishes connections among components through their interfaces and interaction channels [Crnkovic *et al.* 2011]. Binding is also often called *component composition*, which assumes a composition of the components' functions. There is also work on composing, in addition to functions, the components' extra- functional properties.

Sometimes authors refer to *Connections* [D. Perry and Wolf 1992] and distinguish them from bindings. This is generally done in cases where connectors are first-class citizens and when the notion of port also exists. In these cases, bindings become a hierarchical composition mechanism, used between different granularities, while connections are a flat composition mechanism, used among the same granularity. We do not utilize this distinction.

2.3.8 Component framework

Components run in environments that provide them support services, which we call *component frameworks*. Such services may include deployment, automated assembly, communication, third-party binding, scheduling, quality of service, persistence, among many others. To better understand the concept, generally, components are atomic structural elements used to construct an application, and once provided to the framework, they are instantiated and started by said framework. To this extent, frameworks are containers that manage the components (*e.g.*, their lifecycle) and their interactions with other components (*e.g.*, their bindings) and with provided services. As stated, components should make dependencies explicit [Szyperski 1997][Kon and Campbell 2000], although it is often tolerable for the framework to be the only implicit context-dependency a component has. This is accepted because components are made for specific component frameworks, and as such, declaring the framework or its mandatory services as a dependency is redundant.

Frameworks may themselves be seen as components and embedded in other frameworks, giving way to hierarchical frameworks or framework nesting. The open source Java Enterprise Edition JOnAS application server[7] is an example of embedded frameworks. At its core, JOnAS runs on an OSGi framework, yet JOnAS provides containers for various other specifications, namely Enterprise Java Beans versions 2.1 and 3.0, servlets and other Java EE Modules (*e.g.*, jar, war, ear, rar). Application developers can then write Java Bean components that run in the EJB container, while the container itself runs in an OSGi framework. Taking this concept further, Dysoweb[8] is an OSGi container for developing dynamic web applications and is provided as a Web Archive that can be run inside of JOnAS. Thus, we have servlets than run in OSGi, that runs in a servlet container, that runs on JOnAS, that runs on OSGi.

2.3.9 Other concepts

There are other relevant concepts related to Software Architecture that remain to be defined. For instance, the concept of Architecture Style [D. Perry and Wolf 1992] is relevant to define general design decisions about the architectural elements and to emphasize important constraints

[7] http://jonas.ow2.org

[8] http://www.requea.com/xwiki/bin/view/Main/Dysoweb

on the elements and their relationships. Architectural styles are used to represent families of software architecture descriptions that belong to software systems that have something in common: resource types, configuration patterns and constraints [Garlan 2002]. Examples of styles are: event-based, publish-subscribe, blackboard, pipe-and-filter, client-server, object-oriented, etc. Also relevant are the concepts of *property* [Garlan 2002] and *constraint* [Andrade *et al.* 2003] to describe the semantics associated to architectural elements or the restriction of the design, respectively. The reader can refer to [R. N. Taylor *et al.* 2009] to get further details about relevant topics within software architecture.

2.3.9.1 Deployment

Component deployment is a process that enables component integration into the system. Although software deployment may be viewed as the—very large—process of getting software to the point where it is available for active use within a user's computer including releasing, installing, activating, deactivating, adapting, updating, version tracking, uninstalling and retiring, we prefer using a more limited definition of deployment to simply describe the phases of obtaining—commonly downloading—and installing the software (which involves dependency resolution).

Deployment is achieved by means of deployment units. These are units that provide the (binary) code and resources needed to construct and run the application. Deployment units and modules are generally the same entity, which, although not mandatory, is felt as a comfortable best-practice since their commonness eases comprehension of the architecture and its assembly. In general, deployment units and modules are a natural fit together, since they both provide constructs around code structuring and attempt to provide reusability.

2.3.9.2 Architecture Description Languages

Software's structure can be described using an Architecture Description Language (ADL). ADLs have emerged as formal languages to define the architecture of software systems [R. N. Taylor *et al.* 2009; Bass *et al.* 2003; N. Medvidovic and R.N. Taylor 2000]. ADLs facilitate communication and assist in expressing, verifying, and imposing properties upon the software. Unlike programming languages, they tend to be declarative; they describe a system's architecture as a set of components, connectors, bindings and configurations.

There are many ADLs that have been developed, such as Darwin [Jeff Magee and Jeff Kramer 1996], Acme [Garlan *et al.* 2000], Rapide, and Wright [R. J. Allen 1997]. Furthermore, architecture-centric software development tools also exist, including ArchStudio and Acme-Studio. Koala [Van Ommering *et al.* 2000] and Fractal [Bruneton *et al.* 2006] are some of the few ADLs that have been used in practice.

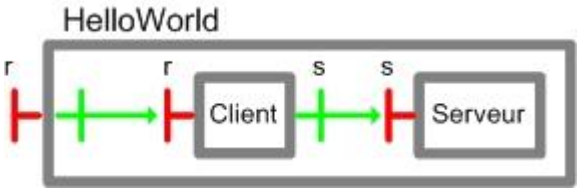

Figure 2: Graphical example of a Fractal Composite component (from Fractal[9])

A graphical example is used to describe a simple Fractal application in Figure 2. The HelloWorld application is composed of two components called *Client* and *Server*. An equivalent application, using an XML syntax is shown in Figure 3.

```
<definition name="HelloWorld">
    <interface name="r" role="server" signature="java.lang.Runnable"/>
    <component name="client">
        <interface name="r" role="server" signature="java.lang.Runnable"/>
        <interface name="s" role="client" signature="Service"/>
        <content class="ClientImpl"/>
    </component>
    <component name="server”>
        <interface name="s" role="server" signature="Service"/>
        <content class="ServerImpl"/>
    </component>
  <binding client="this.r" server="client.r"/>
  <binding client="client.s" server="server.s"/>
</definition>
```

Figure 3: Declarative description of a Fractal Composite component

Fractal components can be composite components or primitive components. Primitive components define their implementation classes (in the case of the Julia of Fractal, implementations are in Java).

In general, ADLs manage complexity by describing hierarchical compositions, which are needed for scalability. In the figure, if a (client) component points to another (server) component, the server component provides an interface, while the client component requires an interface. The lines represent bindings between the components that require and provide interfaces. Fractal provides a structural description of software architecture. However, different systems need to associate functional, behavioral and system properties with the architecture. With the Unified Modeling Language v.2 (UML 2), some ADL proposals are profiles that extend UML by means of stereotypes that extend structural elements with additional properties and constraints.

[9] http://fractal.ow2.org/tutorials/helloworld.html

Although very useful, Software architectures and ADLs are not enough: management and maintenance of these systems still requires great effort [Dewayne E. Perry 2008].

2.3.10 Modules vs. Components

Modules are used to structure code, and as such have generally existed at the programming level (language-level) [Wirth 1977; Wirth 1992]. In modern frameworks, such as in the case of OSGi, this is different because, historically, the Java language has not provided a *module* construct, instead Java provides packages, which are an attempt at structuring the namespace problem (packages are private, protected and public to work as pseudo-modules), originally using reversed internet domain names, although the current trend is to use trademarks. Most propositions for the Java language or Java framework[10] to include have aimed at providing components instead of modules [Aldrich *et al.* 2002] and have not been successful[11]. OSGi, on the other hand, has been very successful but is provided as a dynamic module framework on top of the JVM instead of as a series of language constructs. It has become the de-facto modular Java solution and provides ad-hoc modules and components that exist outside of the JVM and the Java language, avoiding having to alter the Java specification and retaining compatibility across all Java platforms. Although we will better introduce OSGi later on in this dissertation, one could argue against OSGi's solution—since it is fairly large-grained—and say that modularization could be better achieved at a finer grain-level. However, other Virtual Machines, such as .NET, have created similar module-like concepts (Assemblies in the .NET framework[12]) that are not visible in the programming languages but serve many purposes, among them, as deployment units. Furthermore, OSGi is more of a hybrid approach that brings a mix of module, component and service oriented-architecture concepts simultaneously and with a bit of confusion (we will explain OSGi in more detail in Chapter 9).

Modularization is important in that it enhances code reuse, thanks to modules being referenceable from different parts of code or from different applications. Modules are singletons, their definitions exist once and only once in the framework (except for frameworks that support multiple versions of modules that coexist[13], which is generally not the case), there is no notion of instances of modules. The analogy towards operating system libraries is illustrative. Modules, just as shared libraries—which are loaded, initialized and shared among different processes—are only loaded once. Private libraries, being a particular type of module where the module is loaded into a private section of the process's memory, are slightly different since the module is loaded and initialized once for every program that requires it. Even so, it should be noted that this is not the same as an instance, since private libraries are loaded and initialized still only once per-process. Furthermore, it is interesting to see that the developer of the library statically determines if a

[10] It is important to distinguish between the Java language and the Java Virtual Machine. Many languages run on the JVM, both interpreted (*e.g.*, Jython and Groovy) and compiled (*e.g.*, Java and Scala).

[11] Project Jigsaw (proposed for Java 8 http://openjdk.java.net/projects/jigsaw/) is an attempt by Sun, and now Oracle, to provide a standard module system for the JDK. Jigsaw, apparently, would not be visible at the application-level.

[12] http://msdn.microsoft.com/en-us/library/k3677y81(v=vs.71).aspx

[13] For example, Erlang provides modules and can dynamically move new requests from one version of a module to the next [Virding et al. 1996].

library is shared or private beforehand, a user has no say in this. Thus, a library is programmed to be shared or not.

There is confusion as to what constructs may better provide for modules and/or components, whether it be a class or a package, visibility and scoping, super-packages, or some other import/export mechanism, and what constructs should be avoided [Pfister and Szyperski 1998; Szyperski 1992]. Different approaches have been proposed with varying results. When analyzing the propositions, component approaches are common for object-oriented languages (*e.g.*, C#, Java), and modular approaches are mostly proposed for procedural or functional programming languages (*e.g.*, Pascal/Modula, Erlang).

The differences and similarities between components and modules are a matter of discussion. Szyperski mentions that both modules and components are about partitioning software. Furthermore, he establishes the following:

> *Where modules partition the implementation description of software systems, components partition the systems themselves. It is clear that the independent existence of a component implies the existence of an independent description of its implementation. That is, componentizing a system naturally leads to modularized implementation descriptions. The inverse does not generally hold, though. Quite the contrary, blobs of bits are fused from compilation steps in linking stages. Whether the source was modular or not is not relevant, but it has no effect on the fact the blob is at best a single component. Modularization does not necessarily lead to componentization. Whether the source used at construction time was modular or not is relevant for many reasons, but it has no impact on the fact that the resulting blob, is, at best, just a single component. In other words, modularization does not necessarily lead to componentization.*
>
> *[Szyperski 2000]*

In general, a module should encompass a series of cohesive classes and should be loosely coupled to its external dependencies. Modules may require initialization (*i.e.*, an initializer), a single piece of code that is to be executed before using the module, and finalization (*i.e.*, a finalizer), which are to be executed once they are no longer used and before they are removed. Initialization and finalization are commonly used with physical devices.

In practical terms, when using a module an explicit dependency needs to be declared towards it (possibly including the version or range of accepted versions, among other metadata), making a module a code-level construct (albeit in C a module is implicitly created and is formed by the practice of using separate files, no clear module construct exists) and also a deployment construct (think of libraries such as *.dll*, *.a* or *.so* which are loadable modules).

There is reason to have modules and components coexist, each with their clearly defined tasks. In our view, code and resources are contained in modules, which are deployment units that can deployed dynamically onto the framework, and are used to construct the components. A component definition can be spread over many modules, and modules may participate in more than one component definition. Once a component is complete (*i.e.*, all its code is there), it can be instantiated and provides services.

2.4 Service Oriented Computing

Service-oriented computing (SOC) [Papazoglou 2003; Huhns and Singh 2005] is a paradigm that defines a service as the fundamental unit for application design. Services are self-describing components that support composition of distributed applications. Among the objectives of SOC is to define and reduce dependencies between functional units and to promote substitutability. By reducing dependencies, each element can evolve separately, so the resulting application becomes more flexible than monolithic applications. SOC is based on three actors:

- A service provider offers a service.
- A service consumer uses a service.
- A service registry holds services' specifications and references to their servants.

Services are described using a service specification, which is a description of its functionality (*i.e.*, a service interface), and which may include its non-functional characteristics and semantics. A service provider publishes its service specification and the reference to the service implementation using the service registry. Consumers may search for services using the registry and then invoke them once they have a reference to the implementation (called servant in distributed object oriented frameworks). This provides discovery, selection, binding and composition of services. Figure 4 depicts the interactions that take place in a Service-Oriented Architecture (SOA). The service registry is a role always played by the framework, while the service consumer and provider are roles played by components. A subtle difference between the basic SOA and a dynamic SOA concerns the notifications from the service registry towards the service clients. These notifications are independent of the lookup, and may arrive at any time informing of the registration and withdrawal of services. Service clients can thus choose the most suitable service provider dynamically.

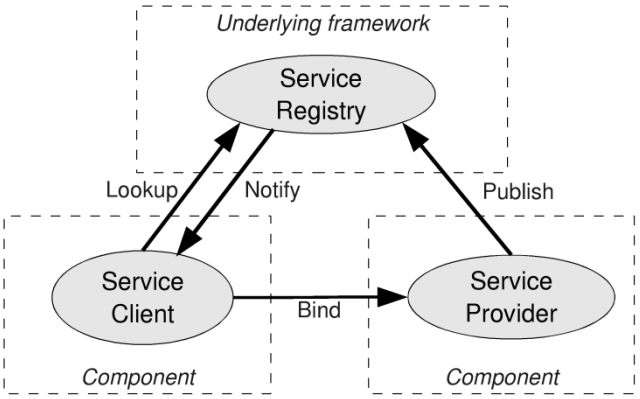

Figure 4: The Service-Oriented Architecture

In general, SOC provides the means to achieve substitutability, which is the basis for dynamism, by supporting the following properties that are exploited for dynamic applications:

- **Loose coupling:** a consumer needs only to know what is specified in the service specification.

- **Late binding:** a consumer may consult the registry at any time to bind to a service implementation.
- **Dynamic resilience:** service consumers do not rely on the same service implementation being returned.
- **Location transparency:** providers and consumers are oblivious of the underlying infrastructure.

In order to build complex service-based applications it is necessary to compose services to provide higher-level services. Furthermore, service providers may require other services in order to operate correctly. This entails service-dependencies, where providers publish their services when their dependencies are met, and they may retreat them when not.

Service-oriented applications require additional attention because of inherent dynamism, making them difficult to implement and error-prone. The complexity involved has led to component-based approaches that use the SOC concepts but advocate the separation-of-concerns principle. The next section describes how SOC concepts are merged into component models to provide dynamically adaptable software systems.

2.5 Service-Oriented Components

As we have seen, CBSE provides us with a *divide-and-conquer* approach to reducing application complexity. By structuring software into modular units with clearly defined roles and interfaces, we facilitate the construction of compatible implementations. Replacing parts of an application comes down to choosing a compatible building block and integrating it. Initial approaches using components defined the architecture of the application at design-time and compile-time introducing tight coupling between components, making it difficult to replace them at runtime. Newer techniques have provided mechanisms for achieving this at runtime but still lack a level of flexibility that the component approach tends to inhibit. One current limitation we find is that when constructing applications we choose a specific implementation and not a desired functionality. Late binding may partially solve this by performing the component bindings at runtime; however, code is still coupled to a given implementation. For example, when choosing a logger for an application, normally we would specify requiring *Log4j* instead of being more general and specifying a *Persistent Logger.* Of course, one could use an additional layer of indirection, perhaps with abstract factories, but if we need to do this for each case, we end up with cumbersome solutions (*e.g.*, factories, abstract factories, multiple approaches). Using the Service-oriented Computing approach, we delegate to a centralized entity decisions regarding finding and instantiating required functionality. The application now becomes specifiable at a higher-level, that of functionality instead of implementations, and this increases decoupling and flexibility. Service-oriented Components are the result of applying the Service-oriented Architecture to component models, bringing more flexibility and increased decoupling.

A component is a software package that encapsulates a set of functions or data. Components can be seen as black-boxes whose functionality is expressed by clearly defined interfaces [Szyperski 1997]. These interfaces are used to connect components for communication and to compose them to provide higher-level functions. The interface acts as the signature for the

component, consumers need only know the interface and can be naive of its implementation. Cervantes [Cervantes and R. Hall 2004] presented the general principles of the *Service-Oriented Component Model*, which we have come to appreciate as an SOA extension to component based development. The proposed principles are the following:

- A service is a provided functionality.
- A service is characterized by a Service Specification that describes its syntax, behavior, and semantics.
- Components implement service specifications.
- The service-oriented interaction pattern is used to resolve service dependencies at runtime.
- Compositions are described using specifications.
- Service specifications provide the basis for substitutability.

The model that results from these principles promotes service substitutability because compositions and dependencies are expressed in terms of specifications. This makes it possible to develop constituent services independently as well as have variant interchangeable implementations. As in SOA, locality is largely irrelevant. In centralized implementations (*i.e.*, single memory space) such as OSGi, a component may provide a service but internally act as a proxy, transparently providing distribution. The selection process for service-oriented components occurs at runtime. Component instances are resolved (and possibly created) by the execution environment and the application starts when the main component's dependencies are satisfied. The service-oriented component model is thus flexible and powerful. Recently, an industrial effort called Service-Component Architecture (SCA)[Marino and Rowley 2010] has been trying to standardize a technology agnostic service component model.

2.5.1 Abstraction levels

Dynamism relies on the service-oriented computing paradigm (*i.e.*, consumer, provider, registry, service specification) to provide substitutability. Depending on the specific implementation technology, concept mappings may vary, but here we provide an overview of implementations using the object-oriented paradigm. Although explained previously in this chapter, we will go over three main concepts:

Deployment unit or module: is used for installing, updating and removing components. A deployment unit provides component types (and other resources) and contains metadata related to dependencies and features.

Component type: is the component specification. It defines the implementation of services and the component's dependencies (by means of service specification dependencies). Because it implements services, it is used to satisfy other components' service dependencies.

Component instances: these are the runtime entities that are composed during execution. A single component-type may be instantiated many times. Components are bound (*i.e.*, bindings) in order to communicate (*i.e.*, invoke services), letting them perform calculations, share data, etc. Component instances are the functional units that implement and provide services at runtime.

2.5.2 Mapping components to objects

Service-oriented component-models are usually written in object-oriented languages. Component abstractions are not natively supported by many platforms (*e.g.*, .Net or Java), so they exist in a more or less transparent manner depending on the underlying framework or language[14], the abstractions being used, and the development model. It is important to visualize component-to-object mappings to better understand the dependencies that exist, which go further than clear-cut service specifications. These mappings become more interesting in centralized component models because they show datatypes and service references that are shared among component instances. There are two concepts we are interested in that affect dependencies: class and object definitions.

Class definitions are the basic unit of design in object-oriented programming. They specify attributes and methods, which make them a mix of data and behavior in an encapsulated entity. Developers are constantly dealing with classes when creating components. They write glue-code classes[15] for binding and assembling components and they directly create their component abstractions by means of classes. Elements from the component model, including the component's business functions, the actors (*i.e.*, consumer, provider, registry), services, specifications, datatypes (including in the specification), are also mapped to their implementations in the object oriented language. The execution platform (*e.g.*, Java, .NET) does not distinguish between a type of object that represents a component, service or data-type, they all consist of the same abstraction.

Object instances are the instantiation of classes. These runtime entities hold the state of the application. There is no mapping that tells us that an object belongs to a specific component instance or component type since these abstractions are generally not reified by the framework.

In Figure 5 we show the abstraction levels that exist in service-oriented component frameworks at runtime, along with their implementation mappings to the object oriented paradigm. The deployment and design levels show higher abstractions and are the views a user will generally work with. At the deployment-level we see modules (*i.e.*, deployment units) on the framework and we can manipulate them, including installation and removal, which are the two basic primitives. Modules contain component types (*i.e.*, component definitions), which are instantiated by the framework to create component instances. Component types and component instances are also commonly reified when a user requires more details at runtime. Component types are in fact a set of class definitions. At the class level, classes inside modules may reference classes from other modules. This is common, for example, for datatypes which are specified in the service specification and shared. These cross-references of classes exist precisely because of data and implementation sharing. At the runtime level, we show object instances and how they reference objects that are defined by classes in different modules. References can be entangled between modules even when we follow a service-oriented computing approach that promotes loose-coupling. The dynamism lies on the runtime view where, although not illustrated here, the service providers could come from different modules and also be replaced during application execution. As a note, we provide an outlined module in runtime and design view (dotted

[14] Some projects such as Darwin[Jeff Magee and Jeff Kramer 1996], ArchWare [Oquendo et al. 2004] or [Odersky and Zenger 2005] have attempted to remedy this.

[15] In newer approaches, containers or frameworks handle much of the glue-code transparently for the developer.

rectangles) as a reference to which module the elements came from, but this abstraction is not actually reified beyond the deployment view. We have also included color coding to show that classes, and hence objects, can later be referenced by other modules which will help understand the issue of dangling objects and static dependencies we will explain later. Furthermore, the combination of the iPOJO component model and the OSGi platform provide similar concepts for centralized application in the Java framework.

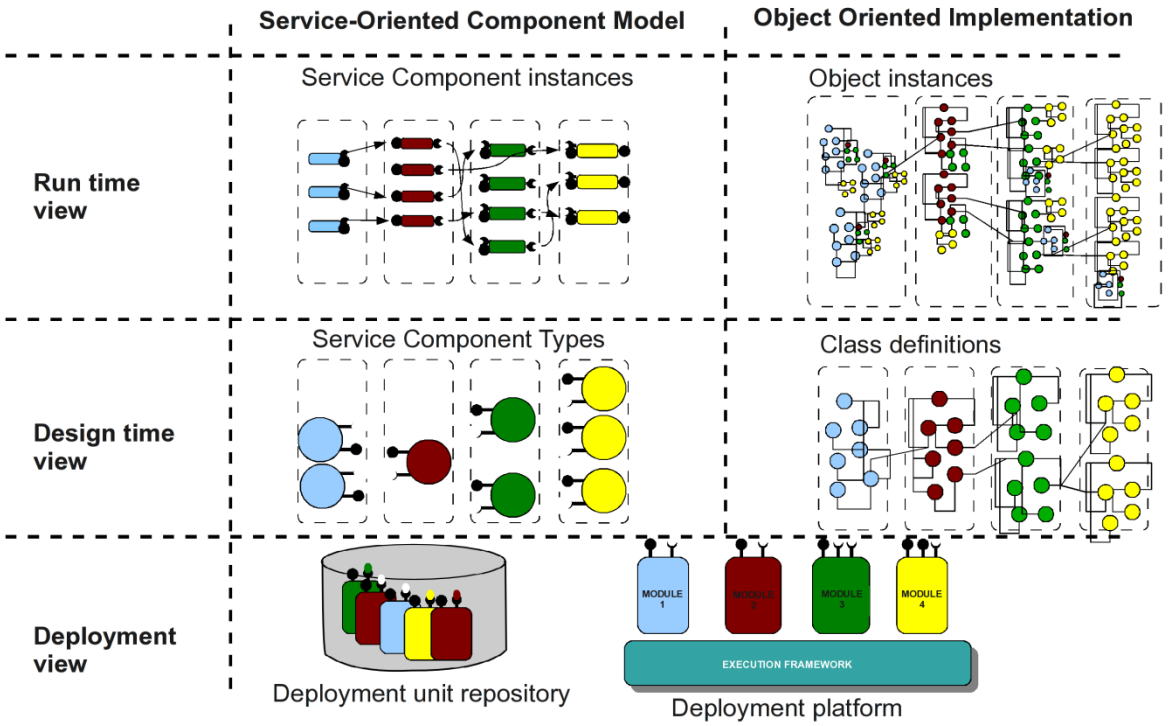

Figure 5: Abstraction levels in service-oriented component model implementations

2.5.3 Dependencies

Dependencies are one of the primary constraints to performing dynamic reconfigurations. Missing dependencies affect the lifecycle of components because they cannot run if their requirements are not satisfied. Implementation code is provided by modules in the form of component types, and the granularity of updates is the module itself. Changing the architecture at a finer grain (that of component instances) is possible, but since no new implementation code is provided, the changes are limited to creating (or destroying) new instances and changing bindings.

For the sake of this work, we have defined the concepts of *implementation dependencies*, which are static, and *service dependencies*, which are dynamic. These concepts will be further developed in our proposition.

2.5.4 Dependency types

Two types of inter-component dependencies have been previously identified for systems that allow loading components at runtime: *prerequisites* and *dynamic dependencies* [Kon and Campbell 2000]. They are similar to our definitions of *static* and *dynamic* dependencies, respectively, concerning their impact on components' lifecycles. In addition, we specify a third level of dependency, which we call *resource dependency*, which is not limited to inter-component dependencies since it may depend on things provided by the environment.

Static dependencies exist when a reconfiguration requires restarting and reinitializing the module, causing its full state to be lost and all its components instances to be destroyed. Because the units of deployment are modules, and implementation dependencies are handled at the module-level, the module is clearly the granularity that is directly affected. State-loss and instance destruction are required when a module imports implementation code from another, and the provider module changes. For example, if module A requires classes from module B, and B is updated, we must also update A to use the new implementation of B. This type of dependency is common for datatypes specified in service specifications and for modules that provide libraries. Implementation dependencies are always mandatory for a module to operate correctly (*i.e.*, they are prerequisites) and are costly because they cause the destruction of dependent modules' component instances (which hold the application's state) when changes are applied.

Dynamic dependencies are those where a reconfiguration is possible without restarting the module and loosing state. These dependencies occur at the service level and benefit directly from the principles of service-oriented computing. Required services may be optional, degrading functionality of client components when not available. Dynamic dependencies affect the component instance and cause rebinding to a compatible service if a change occurs. If no compatible dependency is found and the service is mandatory, then the component instance is stopped, and its provided services removed from the registry until its dependencies can be once again resolved.

Resource dependencies, generally regard configuration, and can be either static or dynamic. For example, a communication port, according to how the component is implemented may be static, and require re-initialization of the module to change, or may be dynamic having the component internally handle the change. Also, a port may not be used by two components simultaneously, so declaring these dependencies helps avoid conflicts at runtime. Other examples include hardware devices and files. In general, these dependencies specify if the resource they require can be shared or not (*e.g.*, a file might be read simultaneously) and if the dependency is static or dynamic. The effects at runtime are the same as for static dependencies if the resource is static and dynamic dependencies if the resource is dynamic.

2.6 Conclusion

We have given a short overview of the background concepts recommended for understanding this dissertation. We have shown that components are an inherent part of software architectures and that there are many concepts—at different abstraction levels—that must be managed when building dynamic applications.

Much has been gained from the move to components. Component based software help us tackle complexity, and to prove it, we can take a look at the size of software and confirm that we have moved from large software being thousands of lines-of-code to being millions and tens of millions of lines-of-code.

Components themselves have also largely benefited from the increased decoupling introduced using services. In general, programming using a service-oriented component model increases application resiliency, substitutability and reusability. This is partly because service-oriented components are developed with fewer assumptions regarding the availability of their dependencies, which may disappear at any time, and because they are developed using common specifications and interfaces that make them easily replaceable.

Moreover, we have started to see if current frameworks sufficiently support the construction of dynamic applications. Unfortunately, there are still many shortcomings in current frameworks in order to properly handle dynamism. As we have seen, it is possible to introduce and tolerate dynamic behavior using service-oriented components, however, this is not done while covering the various abstraction levels simultaneously. Indeed, dynamism is a transversal concern that affects design, source code, packaging, deployment and execution of software. A framework that integrates and handles dynamism across these levels is still missing.

In our mind, a framework for the support of dynamic applications should provide at least the following functionality:

- General suitability. The framework should support single and multi-threaded applications in order to exploit the underlying resources and to provide better reactivity. Components should be allowed to freely create internal threads and allow threads to cross component boundaries. Re-entrant invocations (*i.e.*, cycle calls – calls that include the same component more than once) to components should be allowed, permitting cyclic architectures. Stateless and stateful objects should be permitted. Blocking calls should also be permitted (calls that under some circumstances block the calling thread).
- Correctness. The framework must ensure that dynamism results in a correct system and that corruption does not take place.
- Consistency. The programming model should allow safe-stopping components to ensure consistency and avoid corruption.
- Minimal impact on execution. The framework should maximize availability and minimize the number of components impacted by dynamism. Requests currently running in the application should be interrupted as little as possible. Furthermore, state-loss incurred from component removal should also be avoided when possible.
- Maximum transparency. Application developers should not be burdened with all the intricacies involved with each dynamic reconfiguration. A clear programming model should be established in order to ensure that components follow the minimal requirements for dynamism. Although complete transparency is not possible

because it leads to inconsistencies, the impact on source code can be mitigated. Furthermore, dynamism should be separated from business logic where possible.

- Proactive and reactive change. The framework should tolerate top-down changes that are under its control, as well as bottom-up changes to which it must react.
- Unexpected dynamism. Developers and architects should not be forced to identify all origins of dynamism. On the contrary, they should be allowed to constrain dynamism to certain parts of the software while allowing more flexible changes in others.
- Recovery. Should inconsistencies or corruption be introduced into the system because of dynamism, the framework should automatically recover and re-establish normal system execution and correct behavior. Such examples include unexpected sources of dynamism, such as component failures, disconnected physical devices, failed remote services.
- Impact analysis. Analyze the collateral impact of a dynamic change event (*e.g.*, update, substitute, remove).
- Change impact analysis. The collateral impact, or side-effects, of dynamism (*e.g.*, component update, substitute, removal) must become explicit because a single change to an individual software component will, evidently, affect its dependents, but less clearly it may have cascading effects across the application. Indeed, one component can affect the entire application. The impact of each change should be calculable before it occurs.

Chapter 3

Software Evolution

"The reasonable man adapts himself to the world. The unreasonable man persists in trying to adapt the world to himself. Therefore, all progress depends on the unreasonable man."
—George Bernard Shaw

Evolution is a crucial process for all living creatures. According to Futuyma in his book on evolution [Futuyma 2009], *"evolution is the change over time in one or more inherited traits found in populations of individuals."* The theory of evolution explains that all living organisms have evolved from a common ancestor. The diversification and variations of life are described by Charles Darwin as *"endless forms most beautiful and most wonderful"* [Darwin 1859]. Evolution is the cause for speciation. This has led to an enormous quantity of both, highly specialized and more generalized, yet very diverse species. Speciation happens because living creatures need to adapt, and they adapt because they need to become better suited to their environment. Because all life is under these pressures, there is an effort to adapt quickly and to gain an edge in each environment. The well adapted are more likely to survive, and those who stop adapting or have less beneficial traits perish, which is otherwise known as *Natural Selection*.

Software is not living in the same sense as species. Furthermore, it does not suffer from the decay problems that hardware face. It is intangible in this sense. Yet, software suffers from continuous external pressures to change. The power and logical flexibility of computing systems, the extending technology of computer applications, the ever-evolving hardware, and the pressures for the exploitation of new business opportunities all make demands [Belady and M. Lehman 1976]. Thus, software, much as living creatures, must also evolve or perish.

In this chapter we present the state of the art of software evolution. Software evolution is, in many senses, the precursor to dynamic software evolution, and as such, it is a precursor to building dynamic applications.

3.1 Definitions for software evolution

The Merriam-Webster dictionary defines evolution as *"a process of change in a certain direction"*. Webster's New World Dictionary defines it as a *"process of development, as from a simple to a complex form, or of gradual, progressive change, as in a social and economic structure"*. These definitions are, in each case, very general and can apply to much more than just software. Their

generality lacks precision. Mittermeir mentions that evolution is neither revolution (a complete, pervasive, radical change) nor complete stand-still [Mittermeir 2002], which is important in putting evolution into perspective. It is a more fluid concept, where one can find endemic change.

M. M. Lehman, who focused much of his research on understanding of software evolution, provides the following definition of evolution:

> *"a […] process of discrete, progressive, change over time in the characteristics, attributes, [or] properties of some material or abstract, natural or artificial, entity or system or of a sequence of these [changes]".*
>
> *[Cook et al. 2006]*

Regarding *software evolution*, there is no single accepted definition [Mittermeir 2002]. There have been two main fronts on the study of software evolution, 1) the *what* and *why*, and 2) the *how* [M.M. Lehman 1980]. The former has focused on the properties of evolution, its causes and identification of its drivers. The latter, concerned with *how* evolution works, studies the activities, methods, tools and technology to provide the means to control software change [Meir Lehman and J. Ramil 2001]. Both views are complementary, though the latter is more common. However, to master the technology and justify the deployment of good practice in industrial processes, understanding the *what* and *why* is also important. Such understanding provides insight into achieving the goals of the *how*.

The following table, known as Lehman's *"Laws of Software Evolution"* is a major contribution to identifying the causes and processes of this complex phenomenon. The eight laws are summarized in Table 1. They describe a set of general principles for the evolution of software systems. Their purpose is to capture knowledge about the common features of frequently observed behavior in evolving software systems.

I	Continuing Change (1974)	Systems must be continually adapted else they become progressively less satisfactory
II	Increasing complexity	As a system evolves, its complexity increases unless work is done to maintain or reduce it[16]
III	Self-Regulation (1974)	The evolution process is self-regulating, with a distribution of product and process measures over time that is close to normal
IV	Conservation of Organizational Stability (1980)	The average effective global activity rate in an evolving system is invariant over a product's lifetime
V	Conservation of Familiarity (1980)	During the active life of an evolving system, the average content of successive releases is invariant
VI	Continuing Growth (1980)	The functional content of a system must be continually increased to maintain user satisfaction over its lifetime
VII	Declining Quality (1996)	Stakeholders will perceive to have declining quality unless rigorously maintained and adapted to its changing operational environment
VIII	Feedback System (1974–1996)	The evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable baseline

Table 1: Laws of software evolution adapted and simplified from [M M. Lehman **et al.** 1997]

[16] For example, refactoring is a common method of reducing complexity when systems grow.

Our work has focused on the *how* of software evolution. As such, the rest of this dissertation will be related to the technologies, techniques, tools and methods for achieving software evolution.

3.2 Software Maintenance vs. Software Evolution

Software maintenance and software evolution are often considered to be synonymous with one and other. This is in part, as explained before, due to the lack of consensus regarding their definitions, and, possibly due to the only recent success of software evolution as a discipline in the software engineering community.

Software maintenance was popularized by the Waterfall life-cycle introduced in 1970 by Royce [Royce 1970]. In this process, maintenance is the final phase. Only bug fixes and minor adjustments are supposed to take place during that phase. This view lasted a long time, it even influenced the IEEE 1219 Standard for Software Maintenance [Mamone 1994], which defined software maintenance as:

> *"modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."*

The limitations of the Waterfall process took some years to become apparent. Namely, the model is too strict and inflexible, and it should not be assumed that the requirements are known before starting the software design phase or that they do not continue to change during the software's lifetime [Mens 2008]. Fortunately, newer software processes that increase flexibility have been proposed and are quite popular, such as *extreme programming* [Beck 2000] or *Scrum* [Schwaber and Beedle 2001].

More recently, the ISO/IEC 14764 IEEE Std 14764-2006 defined software maintenance as:

> *The totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage (planning for post-delivery operations, supportability, and logistics) as well as the post-delivery stage (software modification, training, and operating a help desk).*
>
> *[ISO/IEEE 2006]*

Mens argues, from a linguistic point of view, that use of the term software evolution, as opposed to maintenance, is preferred because of the negative connotation of the latter term.

> *"Maintenance seems to indicate that the software itself is deteriorating, which is not the case. It is changes in the environment or user needs that make it necessary to adapt the software."*
>
> *[Mens 2008]*

From an engineering perspective, software evolution encompasses the activities of software maintenance. Maintenance activities focus on keeping a product operational and usable. They correct faults, improve performance and make changes to prevent problems. However, new features are not considered maintenance, they are normally considered evolution. Many projects

use semantic versioning (*e.g.*, Apache[17] or OSGi[18]) to reflect the differences between maintenance activities, such as bug fixes, which change only the minor version number, from important changes such as new features, which are an evolutionary activity and change the major version number. Changing the major version number generally also indicates that the specification for the component is no longer compatible, which implicitly implies evolution.

In short, evolution involves substantial changes at both an individual component-level and at an architectural-level. Maintenance involves minor changes, mostly in individual components. As such, maintenance can be a part of evolution, but not the other way around. Evolution is more general than maintenance. To integrate evolutionary changes into software it is necessary to handle them at an architectural-level.

3.3 Evolution as part of the development process

Several authors have used *software evolution* as the term of preference to refer to the phases starting from the initial creation of the software until its retirement. Several methods, including the Staged model [Bennett and Rajlich 2000] and Agile Software Development [Schwaber and Beedle 2001] have been introduced that consider evolution as an important process to software development.

3.4 Evolution and System Architecture

The relationship between evolution and architecture is an interesting one, since changing an architecture can influence the overall system, making it evolve. The IEEE Standard 1471–2000 provides an interesting definition of architecture that mentions evolution:

> *"The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution"*
>
> *[M.W. Maier et al. 2001]*

Thus every system has architectural properties, which may be deliberate or accidental. In either case, they crystallize assumptions about the expected evolution of the system. However, the evolution that actually occurs may not be what the designers of a system's architecture were expecting at the time when architectural choices were made. Whenever a system's architecture incorporates assumptions about the real world that no longer hold and the discrepancy cannot be overlooked, then the system's stakeholders may be faced with either replacing or re-architecting the system. If a software system models a real-world domain, there will always be a risk that this situation could arise.

[17] http://commons.apache.org/releases/versioning.html
[18] http://www.osgi.org/wiki/uploads/Links/SemanticVersioning.pdf

3.5 Conclusion

Software evolution is a very large discipline of study, that, as of recently, covers much of the software development process. There are two general groups of research on software evolution, the *what and why*'s and the *how*'s. Although our interest is in the mechanisms and tools to achieving software evolution, the why we evolve software is important in providing a solution that is practical, usable and feasible.

We provide our own definition of software evolution:

> *Software evolution is the activities that adapt software by correcting, improving, extending or reducing its functionality to satisfy the ever-changing requirements established by its users and by its environment.*

Finally, if we accept the fact that software architecture and software evolution are strongly linked, taking dynamism into account at the architectural level is a natural step in the process of supporting software evolution. We conclude that architects should define the application's architecture with dynamism in mind. The design environment should support the architect by calculating the extent the proposed architecture will be capable of accommodating dynamic change, and conversely, assist the architect in finding ways to better resist and accommodate dynamic change. Clearly such support does not exist today and is an important objective of our work.

Chapter 4

Dynamic Software Evolution

What's software architecture?
"It's the stuff that will be hard to change"
—Martin Fowler

Many systems are deployed by companies that require them to be constantly available. Halting software to make changes, such as updates, bug fixes, add new features, or other enhancements cannot be afforded. Such services may include life-critical systems, financial systems, telecommunications, and air traffic control, among many others. Therefore, techniques are needed to change software while it is running. This is a very challenging problem and is known under a variety of terms, such as, but not limited to, runtime evolution, runtime reconfiguration, dynamic adaptation, dynamic upgrading, hot updating and, our preferred term, dynamic evolution.

However, updating the executable code is the last step of software evolution. As we saw before, all artifacts produced during the entire life of a software are subject to change, including source code, requirements and executable code. In this dissertation, the focus will remain on adapting a system's architecture and applying those changes at runtime.

4.1 From code to execution

To help understand where dynamic software evolution occurs, it helps to understand the processes that are necessary for the construction of software systems. Software is organized into a set of *modular units*, be it classes, components, files, procedures or so forth. The source code that composes these units is compiled into a target language (often binary) by a *compiler* for it to be executed. Separate files are generated (*i.e.*, separate compilation) for each module (*e.g.*, Java class files, C object files), which a *linker* can then use to construct a final executable binary or a library (*e.g.*, .dll, .so, .a) if desired. Each module header has a symbol table with information that defines its dependencies (*e.g.*, shared libraries).

When running an application, the process of dynamic linking takes place. This is in fact different from the previous linking phase, a *dynamic linker* is a special loader that loads external shared libraries into a running process and then binds the shared libraries dynamically to the running process. Dynamic linking is operating system dependent.

In the case of interpreted languages the compilation phase does not occur, an interpreter program reads and directly executes the source code. Some hybrid systems, such as Java and .NET, use a mix of compilation and interpretation. The languages are compiled to an intermediate format (*e.g.*, bytecode), which is then interpreted by a virtual machine. Furthermore, in Java classloading is lazy, *i.e.*, classes are loaded only when needed and not earlier.

In this environment, to introduce changes to a program, first, the source code must be edited and then recompiled. The binary code generated must be re-linked to generate a new executable. The old executable program must be stopped in order to start the new executable, causing application downtime. This is necessary because the new binary has changed symbols and references. This process is known as *offline* or *static evolution* because the system is restarted.

Static evolution causes the current state of the program to be lost, as well as all open transactions. In addition, the application becomes unavailable during the time it stops and restarts. Some solutions have attempted to address this issue by using redundancy (another application takes on the load) and performing soft-restarts to avoid current transaction losses (where the application stops accepting new transactions but finishes current ones). Other research has addressed the fundamental issue of introducing new changes at runtime.

The phases we have explained have been categorized according to when software changes are applied. At least three categories are apparent, based on when the change specified is incorporated into the software system. Specifically these are:

- Static. The software change concerns the source code of the system. Consequently, the software needs to be recompiled for the changes to become available.
- Load-time. The software change occurs while software elements are loaded into an executable system.
- Dynamic. The software change occurs during execution of the software.

In this dissertation we will be focusing on dynamic changes, *i.e.*, changes applied at runtime.

4.2 Introducing changes at runtime

R.S. Fabry introduced the need for updating systems at runtime [Fabry 1976]. In his article, he mentions the cases where updating a module is easy (*e.g.*, the modules does not have permanent data structures and the interface does not change), to those where updating is more complicated (*e.g.*, the module uses permanent data structures) and he provides a solution to the latter. He describes updating at runtime as: "*constructing a system in such a way that the programs and the data structures which they manage can be changed without stopping the system*". Fabry coined the process as *on-the-fly program modification*.

Other terms have emerged since the work of Fabry, yet their goal (and sometimes their solution) is similar. To mention a few, there is *dynamic program updating* [Segal and Frieder 1988], *dynamic change management* [J. Kramer and J. Magee 1990], *on-line version change* [Gupta *et al.* 1996],

runtime evolution [P. Oreizy *et al.* 1998], *dynamic evolution* [Malabarba *et al.* 2000], *live updating* [Y Vandewoude and Y Berbers 2004], *or online evolution* [Q. Wang *et al.* 2006].

4.3 Dynamic software evolution definitions

Kramer and Magee provided, albeit in an informal way, in their article that introduces quiescence, a definition for what we consider to be dynamic evolution:

> *"[Evolutionary change] may involve modifications or extensions to the system which were not envisaged at design time. Furthermore, in many application domains there is a requirement that the system accommodate such change dynamically, without stopping or disturbing the operation of those parts of the system unaffected by the change."*
>
> *[J. Kramer and J. Magee 1990]*

Their view on evolution gives us insight. They see evolutionary changes as unpredictable or unforeseeable at design time. Furthermore, they place emphasis on the need to apply changes while minimizing disruption.

Wang provides a clear description of what dynamic evolution is:

> *"Online software evolution is a kind of software evolution that updates running programs without interruption of their execution."*
>
> *[Q. Wang et al. 2006]*

They use the keywords "running programs" and "without interruption". Running programs are the target software systems, while no interruption is the constraint from the end user. They explain that during an evolutionary process, requests should not be refused or canceled, but the quality of service may decline a little.

Although there are other definitions, by many such authors, most of them are lacking, either by being too specific or by being too abstract and not addressing the issues at hand. Other definitions relating to dynamic updating or live updating tend to gravitate around updating single modules at a low level. It has been argued that approaches like these, focusing on *programming-in-the-small* [DeRemer and Kron 1975], are too low a level, too detailed and impractical due to tight coupling of program elements. Instead, approaches at the component-level (*programming-in-the-large*) should be used [J. Kramer and J. Magee 1990]. We provide the following definition for dynamic evolution:

> *Dynamic software evolution is a guided continuous process of change that enhances, improves, extends or reduces software's functionality in order to satisfy its objective, and is performed during execution, while minimizing the perceived impact of service interruption.*

In this definition, we would like to make evident a couple of terms we have chosen.

- Evolution should be guided; there should be an objective to the changes that the software should attempt to attain.

- Evolution is a continuous process of change.
- Dynamic evolution is performed on a running application, without stopping it.
- The impact of changes should be minimized.

4.4 Dynamic software evolution characteristics

There are many different attributes or characteristics regarding dynamic evolution. We have touched upon a couple of them, such as when evolution may occur (*e.g.*, runtime). Buckley *et al.* [Buckley *et al.* 2005] have proposed a taxonomy of software change. Their taxonomy is focused on the larger picture of software evolution, and as such is not exclusive to dynamic software evolution. We will focus on characteristics of interest to dynamic evolution and specifically to our work, and we invite the interested reader to lookup their work for more detail.

Specialized fields have emerged in dynamic evolution that focus on particular attributes. These fields may be classified by the granularity of their changes (*dynamic reconfiguration and dynamic updating*), and by the activeness of such changes (*reactive evolution, programmed proactive evolution, non-programmed proactive evolution*).

4.4.1 Granularity of changes

The scale of the artifacts to be changed is known as granularity, and can range from coarse through medium, to a very fine degree. Traditionally, many researchers have distinguished only between coarse-grained and fine-grained artifacts with the boundary specified as being at file level. Anything smaller than a file is generally considered a fine-grained artifact [Buckley *et al.* 2005]. Furthermore, most solutions only provide changes at a single granularity-level, leaving open the possible exploitation of providing fine-grained and coarse-grained changes.

- **Coarse-grain changes** are changes to a system's architecture. These changes may impact large subsystems or the entire software system. *Dynamic reconfiguration* is a field that has focused on architecture changes at runtime, by adding and removing components and bindings.
- **Medium-coarse changes** affect component compositions, modules, classes and all of their instantiations. *Dynamic Type Evolution* is a field that addresses this type of dynamic evolution. It provides the modification of types at runtime.
- **Fine-grain changes** are applied to individual variables, functions or statements. *Dynamic updating* [Hicks 2001], which are generally language based solutions, focus on providing these features.

It is noteworthy to mention that higher granularity levels imply changes at lower levels, since, for example, changing a component implies changing its classes and inner variables. The inverse is not true; a fine-grained change does not imply that the architecture is different.

4.4.2 Activeness of change

Software systems can be *reactive* (changes are driven externally) or *proactive* (the system autonomously drives changes to itself) [Buckley *et al.* 2005].

A **reactive** system's changes are driven externally; the system responds to events initiated somewhere else (*e.g.*, a user interface). Support for reactive change enables unforeseen changes (changes not initially predicted during the design of a system).

A **proactive** system must typically contain monitors (*e.g.*, sensors and actuators) and some logic for self-change based on the information the monitors receive. There are two types of proactive changes:

- **Programmed evolution:** changes are designed into the system and are activated when an event occurs or a condition becomes true. This approach has been applied to software architecture and is known as programmed reconfiguration [Endler and Wei 1992], which uses architecture specifications to determine when a reconfiguration occurs and what must change. Many projects have taken this approach [Bradbury *et al.* 2004].
- **Non-programmed evolution:** changes are automatically created at runtime by the system, which decides when to apply them. This is the most difficult type of change to evolve systems, but it is also the most powerful. Few works of research have addressed this type of change. An example is the work of Sykes on generating reconfiguration tasks from high-level goals using a double control-loop [Sykes *et al.* 2008].

Both types of change, reactive and proactive, should be supported in order to introduce unforeseen changes and to reconfigure autonomously. They are complementary and increase the flexibility of the system.

4.5 Dynamic software evolution issues

Many (difficult) issues must be handled when dealing with dynamic software evolution. Two issues that are of importance to this work are:

1. How to deal with stopping artifacts that need to be changed while leaving the system in a consistent state; and
2. Transferring state from stopped artifacts to new artifacts.

The first problem deals with maintaining **application consistency** while minimizing the impact of changes to the system; and the second one, **state transfer**, refers to the migration and/or transformation of the internal structure of data and information (otherwise known as *state*) of an artifact at runtime.

Although many of the concepts presented in this section refer initially to distributed systems, they are applicable to component-based systems, either centralized or distributed.

4.5.1 Safe stopping of running systems

It is a crucial property to leave systems in a consistent state after a change is performed. Detecting when it is safe to actually change the software system is key to ensuring a consistent state. Furthermore, an executing system will potentially have requests that are being serviced at the time the change is to be applied. These requests should preferably not require cancelation, and changing the individual artifacts should not lead the application to an inconsistent state (such as removing an active artifact). For this reason, affected artifacts should be put into a *safe state* that guarantees no inconsistencies will be introduced into the system. In addition, it is desirable to do this with as little disturbance to the system as possible. In doing so, the primary design concern is to find the right trade-off between minimal application disruption and timely exchange.

In this section we present the various approaches for achieve safe reconfigurations, including the main two methods: *quiescence* and *tranquility*.

4.5.1.1 Quiescence

Quiescence is one of the most influential criterion for safe stopping. It was proposed by Jeff Kramer and Jeff Magee in [J. Kramer and J. Magee 1990], where they presented a number of requirements to ensure consistency when reconfiguring a distributed system. They represent the distributed system as a graph, where *nodes* are processing entities and *edges* between nodes indicate communication channels, *i.e.*, a node can initiate a *transaction* by invoking another node, and transactions are the only way to affect another nodes state.

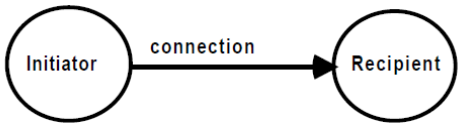

Figure 6: A connection

A transaction is an exchange of information initiated by one node that occurs between two and only two nodes, as shown in Figure 6. The node that starts a transaction is the *initiator*. Transactions consist of a sequence of one or more message exchanges between the two connected nodes. They assume that transactions complete in bounded time and that the initiators are aware of their completion. Figure 7 shows two transactions, `a` and `b` (note that `b` is composed of multiple calls), between two parties.

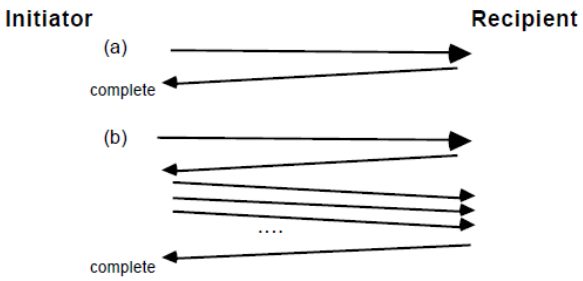

Figure 7: Examples of two-party transactions.

There's a distinction between *independent* and *dependent* transactions. A transaction *t* is said to be dependent on another transaction *u* if *t* can only complete after *u* has completed. It is said that *u* is a *consequent* transaction of *t*.

Kramer and Magee abstract the status[19] of an application into a set of different configuration statuses for each node and consider two main statuses for each node, active and passive, whose definitions are given as follows:

- **Active status:** a node in the active status can initiate, accept, and service transactions.
- **Passive status:** a node in the passive status must continue to accept and service transactions, but:
    1. it is not currently engaged in a transaction that it initiated and
    2. it will not initiate new transactions.

A passive status is a necessary but insufficient condition for updatability because a node may still be processing transactions that were initiated by other nodes. Therefore, they introduce a stronger concept:

- **Quiescent status:** a node has a quiescent status if:
    1. it is passive (it is not currently engaged in a transaction that it initiated, it will not initiate new transactions),
    2. it is not currently engaged in servicing a transaction, and
    3. no transactions have been or will be initiated by other nodes that require service from this node.

In the quiescent status, a node is both *consistent* and *frozen*. It is consistent in that the node does not contain partially completed transactions, and is frozen in that the application state will not change as a result of new transactions.

To change a node Q to a quiescent status we must ensure that no transactions have or will be initiated by nodes that require Q. This implies that the following nodes will have to be passivated also:

- Node Q
- All nodes which can directly initiate transactions on Q, *i.e.* all nodes with connection arcs directed towards Q
- All nodes which can initiate transactions which result in consequent transactions on Q.

Kramer and Magee defined this as the enlarged passive set (EPS) of a node Q, denoted *EPS(Q)*. They demonstrated that, in a system with nested transactions and assuming that these transactions complete in bounded time, a node Q can move towards the quiescent status in bounded time if all the nodes in EPS(Q) are passivated.

Quiescence is sufficient to ensure consistency and it is reachable in finite time (as long as individual transactions complete in finite time). Quiescence was implemented in the Conic environment for distributed programming [J. Magee *et al.* 1989] and has since become the basis for

[19] Kramer and Magee use the term state instead. As proposed by [Yves Vandewoude et al. 2007] we use the term status to distinguish the internal state of a node from its relation to the evolutionary process.

many other systems. Nevertheless, achieving quiescence in a system (with nested transactions) is stringent and invasive and often causes serious disruption [Yves Vandewoude *et al.* 2007]. This is because every node that may directly or indirectly cause a node Q to service a transaction must be passivated for Q to be stopped (and eventually replaced). A second important drawback is that quiescence breaks black-box design because it assumes that a node has knowledge of whether its actions are part of a transaction initiated by another node. This increases coupling and hinders reusability.

4.5.1.2 Tranquility

Although quiescence is a sufficient condition for the ability to update running components, it has a large drawback regarding the impact on the system when enforcing it. It is not sufficient for the node to be updated to be simply put into a passive state, but all nodes that are directly or indirectly capable of initiating transactions on this node must also be passivated. This is a serious drawback because of the potential impact a change can have on the system, possibly bringing it to a halt. Vandewoude *et al.* addressed this problem in 2006 when they introduced the tranquility criterion [Yves Vandewoude *et al.* 2006].

Tranquility[20] reduces the constraints that quiescence exhibits but relies on two basic assumptions:

1. the original and resulting configuration are valid, and
2. each node should only rely on external functionality.

Tranquility is easier to obtain and less disruptive than quiescence, and still sufficient to ensure consistency before changes. Tranquility exploits black-box design. Tranquility is, by definition [Yves Vandewoude *et al.* 2007]:

Tranquil status (tranquility): a node is in a tranquil status if:

- it is passive (it is not currently engaged in a transaction that it initiated, it will not initiate new transactions)
- it is not actively processing a request
- none of its adjacent nodes are engaged in a transaction in which this node has already participated and might still participate in the future

To explain what this means, a node that participates in an active transaction can be safely replaced if:

- it has finished
- it has not yet begun
- it is part of a sub-transaction

In replacement operations, which are the basis for updating software, new transactions that have not yet begun may be executed by the new version of the component (thanks to the black-box

[20] In retrospect, Vandewoude et al. mention that a better name would have been *Latency* because the tranquility condition is not stable by itself, when a node is tranquil all further interactions need to be blocked.

design principles where implementation details are not known and components rely on external, public functionality).

Tranquility is an improvement to quiescence in many cases. Tranquility relaxes some of the constraints defined by quiescence, which effectively reduces the number of nodes that need to be passivated. By definition, quiescence implies tranquility because quiescence will passivate many more nodes than tranquility (including all nodes that tranquility would passivate). Yet the inverse is not true, tranquility does not imply quiescence because tranquility passivates less nodes. Tranquility also adheres to black-box designs, which improve component reusability.

A disadvantage of tranquility is that it does not ensure that the status will be reached in bounded time. Nevertheless, in practical situations this rarely occurs. In addition, tranquility can fallback to quiescence in these cases (for example, after a timeout) to ensure the node is passivated and the system safe and consistent.

Another drawback is the simplified handling of sub-transactions. Tranquility assumes that sub-transactions are inherently independent of the enclosing transaction because they are made independently of the initiator. This independence is not always true. A sub-transaction may use information that has been calculated by the original transaction, so the sub-transaction is not independent of its predecessor. For instance, suppose that a node X, at time t_i has finished its participation in an ongoing transaction, and as a result of this participation, the internal state of X has the value v. Then, at time t_{i+1} another node Y, which is engaged in the active transaction, starts a sub-transaction which involves X and changes its state to v'. According to the tranquillity condition, the node X could be replaced at time t_i by a new version. However, if the new version does not migrate correctly the previous state v, an inconsistency with Y may be produced.

Given these limitations of tranquility, it is important that quiescence be given as a fallback.

4.5.1.3 Other approaches for Safe Stopping

Several solutions for safe stopping and updating components for the Common Object Request Broker Architecture, better known as CORBA, have been provided. Bidan *et al.* provide an algorithm for performing consistent dynamic reconfiguration of CORBA applications, where consistency refers to RPC integrity [Bidan *et al.* 1998]. They passivate links between nodes of a distributed system instead of the nodes themselves, which causes the activities that use them to block, but not the nodes themselves (this allows multi-threaded applications to continue to execute on threads that are not blocked on communication). The impact on execution is minimal. The disadvantage of their approach is that RPC requests must be independent, nested RPCs are not allowed and the reconfiguration of systems with re-entrant invocations is not supported.

Some authors have provided an invasive approach to safe stopping and updating. They propose that developers define locations in the code where changes may occur [Hicks 2001]. Duquesne *et al.* [Duquesne and Bryce 2008] proposed updating by means of loading a whole program twice then rerouting between the old version to the new one. They claim that initialization procedures and general program coherency are not maintained when updating programs by smaller blocks, thus, they provide language constructs for developers to specify the

points where the rerouting should occur. Their solution was limited to single threaded applications and it is difficult to specify a single execution point where an update should occur.

The K42 Operating System detects quiescent states using a mechanism similar to read copy update (RCU) in Linux [McKenney and Slingwine 1998]. This technique makes use of the fact that each system request is serviced by a new kernel thread, and that all kernel threads are short-lived and non-blocking. Each thread in K42 belongs to a generation, which was the active generation when it was created. A count is maintained of the number of live threads in each generation, and by advancing the generation and waiting for the previous generations' counters to reach zero, it is possible to determine when all threads that existed on a processor at a specific instance in time have terminated and as such, when it is possible to perform a safe-update.

The Fractal component model [Bruneton *et al.* 2006] uses interception points in the component membrane to replace a component. The interceptors halt new incoming calls and wait until current calls finish. They use one of two methods to determine when a component has finished open calls, either thread counting or a thread generation technique similar to K42. Once a component has finished all open calls, it can be replaced. Although Fractal and K42 call this quiescence, it does not follow the definition of quiescence given by Kramer and Magee because interception is used instead of passivating other components. This reduces the impact on the system, since less components are stopped, but adds the cost of the interceptors. This is not a proper solution to quiescence because it does not handle transactions (*e.g.*, a series of calls) and does not ensure a stable architecture for each transaction. Furthermore, intercepting open transactions before they can finish, without regard to which components they still require to finish, can leave the application in an inconsistent or potentially corrupt state.

Gomaa and Hussein [Gomaa and Hussein 2004] introduced a set of design patterns for dynamically reconfigurable systems, most of which are based on the concept of quiescence. The contribution of their work is that they specify the behavior required to reconfigure different architectural styles: master/slave, client/server, centralized and decentralized architectures.

Pissius & Coulson have implemented quiescence in the OpenCom component model [Pissias and Coulson 2008]. Their design is based on interception in connectors and uses metadata and interception to obtain information about nodes that participate in ongoing transactions. However, they impose that developers label operations as blocking or unblocking.

The iPOJO component model [Escoffier *et al.* 2007] uses the Inversion of Control (IoC) design pattern so that the component membrane can be notified of calls to components. The membrane uses thread-local variables to store the component's current dependencies. This can be seen as a dependency snapshot for the current thread. It differs from other approaches because the programming model for iPOJO is inherently dynamic, forcing programmers to be defensive against dynamic events, making it possible for a component's dependencies to change from one call to the next. Even threads that call a component in succession may obtain different dependencies for each call to the same component instance. Thus, for each call to a component, and on each different thread, a snapshot of the dependencies of that component is taken and stored in the thread's local variables (*i.e., ThreadLocal*). When the call finishes the variables are cleaned up. This is not a proper solution to quiescence because it does not handle transactions (*e.g.*, a series of calls) and does not ensure a stable architecture for each transaction, but it provides

inherent dynamism to the system and facilitates programming dynamic components. Furthermore, it places a large burden on developers because the programming model must be closely followed, and internal state must be carefully handled, in order to ensure the application is not rendered inconsistent because of dynamic changes.

4.5.2 Handling stateful artifacts

Another key issue when facing dynamic evolution is that of handling *stateful* components. Statefulness refers to the existence of internal state that may change as a consequence of interactions with external elements or spontaneously due to proactive behavior [Hammer 2009]. When replacing components, updating requires that the state of the old component be moved or *transferred*, and possibly transformed, to that of the new component. In the case of stateless artifacts, updating is much simpler since state transfer is no longer necessary (by definition they do not have state). In this case, once a stateless component has simply been quiesced, it can be stopped, removed and replaced by another component.

State transfer is the process of extracting the runtime state of an element and using this information to initialize a new version. If a very fine-grain approach to updates is used, when an update occurs in the middle of a procedure, information such as the CPU registers, the stack, the location in the method, variables, and other information would have to be preserved. Persistent state not related to the activeness of a method would also be required, such as global and instance variable have to be saved. One reason to push for a safe state (explained in the previous section), is that it minimizes the amount of control state that needs to be preserved. This is highly advantageous since it minimizes the complexity of moving from one active part of code to another, while migrating large amounts of small but important details.

In many cases, such as in component models and when using tranquility or quiescence to attain a safe state, the real challenge is not to *copy* the data from one component to the next, it is more precisely to *adapt* or *transform* this data to an acceptable format for the new component. *State transformation* is necessary when data structures change between two components, and is generally highly application specific. Proposing a generic method for state transformation is challenging, yet it is not a central idea of this dissertation. Nevertheless, this is an important requirement to dynamic evolution, and as such, we will explain the three principal methods of achieving it.

No State Transfer. The complexity of state transfer and transformation has led to, what Hjálmtýsson and Gray call *passive partitioning* [Hjálmtýsson and Gray 1998; Malabarba *et al.* 2000], which consists of allowing multiple versions of instances to coexist in the framework simultaneously, where code and values of old versions are used exclusively by older parts of the system, while new instances are always of the newest version. The decision of which to invoke, either old or new, is handled automatically. This is an efficient way to handle updates but the drawbacks become apparent and burdensome to programmers when handling many versions of code and state interacting together and trying to ensure program consistency [Hicks and Nettles 2005]. These defects are very visible in the language-based approaches in which they are used, and they are likely mitigated in a component environment with clearly defined interfaces.

Delegated State Transfer. This approach puts the burden on the developer for handling differences in state. The state is transferred automatically but is not transformed. Transformations

are provided from programmers who have knowledge on the semantics of the state in both versions of a component. This is called a *transformation function*. Some approaches assist the developer in creating transformation functions, as, for example, Hicks *et al.* who provide automatically generated method templates [Hicks and Nettles 2005].

There are two general variants to this approach, namely, *Global Update* and *Active Partitioning*. The former migrates all instances to the new version ensuring that there is only one version available at any single time. The latter allows the programmer to specify which instances are migrated and which continue to use their current version, making it possible to execute with multiple versions simultaneously. Hjálmtýsson and Gray are an example of this approach, they do not convert instances automatically but let programmer may explicitly do so if desired [Hjálmtýsson and Gray 1998].

Automated State Transfer. In order to fully automate state transfer, the dynamic update mechanism must use state information from old and new versions and automatically transform the state from one to the other. This process is complicated because of state semantics (or more precisely the lack of explicitly declared semantics). Another problem emerges when the new version's data structure contains elements that are not available in the old version, such that, they are impossible to fill in with the necessary values (default values are often used). The theoretical limitations of state transfer have been discussed in [Bloom and Day 1993].

Some approaches that have been proposed provide a semi-automatic state transfer mechanism. DeepCompare [Yves Vandewoude and Yolande Berbers 2005] consists of static analysis that takes place and automatically detects similarities in the source code of the new and old component versions, with minimal user assistance, and maps the detected structures between them. The approach was demonstrated to be effective, automatically detecting the transformations in 95% of the cases, but it does require access to source code of both versions.

4.6 Approaches related to dynamic evolution

There are many research fields that have addressed issues related to dynamic change management. As expressed in Chapter 3, software evolution was not considered a serious discipline until many years after the pioneering work of Lehman [M.M. Lehman 1980] and others. This, and the growing needs to have particular aspects of evolution addressed, led rise to the existence of various approaches, more or less specialized to a particular domain. The fields we will address in this section, namely, *Control Systems* and *Computational Reflection*, are all relevant because they address the adaptation of physical devices, the ability for a computer program to observe and change itself, and the configuration and self-management of information technology systems, respectively. Furthermore, these fields share the general notion of control-loops to gather information and adapt the system accordingly and continuously.

4.6.1 Control Systems

Control systems are devices that direct or regulate the behavior of other devices or systems. A control system is thus a process that supervises execution and adapts it, by means of varying different input parameters, in reaction to a stimulus (or stimuli) it receives. There are many

variants of control systems, but they fall into two basic categories, namely *closed-loop* and *open-loop* control systems, depending on where the stimuli comes from.

Open-loop control systems do not make use of feedback, and run only in pre-arranged ways. They compute input using only the current state of the system and a model. These systems are useful where state can be modeled by a mathematical formula. These systems are often used in simple processes of their low cost and simplicity to construct, especially in systems where feedback is not critical. Such examples include a washing machine, where the time to wash is provided by the user, or a lawn sprinkler, which starts and stops at precise times. In the sprinklers case, if it includes a moisture sensor to know if the lawn needed irrigation, for example if it rains it would no longer need to irrigate, it is a closed-loop control system.

In the case of closed-loop, or feedback systems, a control loop—as shown in Figure 8—uses sensors, control algorithms and actuators in an arranged manner to regulate a variable at a reference value. An example is a car's auto-cruise controller, which may increase (or reduce) the gas and air mixture to a motor in order to accelerate (or decelerate) when speed increases (or decreases) beyond a respective threshold-speed. Closed-loop control systems are as such adaptive, *i.e.*, they use sensing in order to adapt to varying circumstances. Other examples may include robots that adapt their speed, refrigerators, air conditioning and heating.

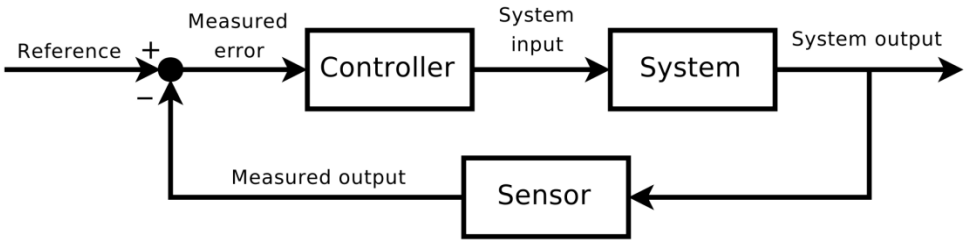

Figure 8: The feedback loop to control the system's dynamic behavior.

Morrison is not the first to show us that these concepts can be applicable to software systems and, more specifically, to the development of adaptive and reconfigurable systems [Morrison *et al.* 2007]. Software is more malleable than hardware, providing it with many benefits in regards to change and adaptation. In the case of software architecture, the architecture itself is the controlled process, while the controllers are the mechanisms that change the architecture dynamically. These ideas, and specifically that of a continuous control-loop that acts and reacts on the architecture, have influenced this dissertation.

4.6.2 Computational Reflection

Reflection is the capability of a program to observe and modify its own structure and behavior at runtime. Although the concept has extended to many fields in computer science, it was initially provided in programming languages. In computer architectures, program instructions are generally stored as data; the distinction between instructions and data being a simple matter of how they are interpreted and treated. The processor executes instructions, while data is read, processed and written. Yet, some languages provide facilities to treat instructions as data and modify them, changing a program's behavior and structure dynamically. This concept arises in the

early 1980s with Brian Smith's doctoral thesis [Smith 1982]. He proposed using reflection in programming languages and worked on variants of Lisp to achieve this.

The usefulness of reflection and its growing popularity helped it spread to many fields, such as distributed computing, operating systems, middleware [Kon *et al.* 2002], component-oriented programming [Bruneton *et al.* 2006], software architectures [Tisato *et al.* 2001], among others. In object-oriented languages such as Java, reflection allows both inspecting and invoking at runtime elements unknown at compile time. You can instantiate classes or invoke methods, inspect the classes or interfaces, recover field names and methods, among other things. Reflection can be used to adapt a given program to different situations dynamically but it usually requires additional knowledge in order to take advantage of more generic code execution. This level of adaptation is achieved thanks to a reduction of hard-coding solutions. In component models, reflection allows a program to access components, instantiate or destroy them. It also allows to inspect components, their interfaces and bindings, and sometimes internal elements such as their data or the classes that are used to implement them. Reflective programming, specifically for reconfigurations, can become complicated and the APIs used do not generally provide many guarantees when one wishes to change an application. Higher-level approaches have been proposed, such as FScript. FPath is an XPath like language designed to express queries on the Fractal component model. FScript [David *et al.* 2008] is a domain specific language that uses FPath to specify dynamic reconfigurations for the Fractal component model. The language separates reflection, and more specifically, reconfiguration from the application itself and provides guarantees to reconfigurations, such as atomicity, consistency and termination.

To better understand the concept of Computational Reflection, Maes gave us an early definition to reflection. Her definition attempts to be general for all computational systems although her work focused on the application of these concepts to object-oriented languages. The definition is as follows:

> *Computational Reflection is the activity performed by a computational system when doing computation about (and by that possibly affecting) its own computation.*
>
> *[Maes 1987]*

A definition of reflection given by Malenfant *et al.* in 1996, in a paper they wrote on the generalities and technicalities of efficiently implementing reflection in programming languages, be them functional, object oriented or even logic programming, is the following:

> *Reflection is the integral ability for a program to observe or change its own code as well as all aspects of its programming language (syntax, semantics, or implementation), even at run-time. A programming language is said to be reflective when it provides its programs with (full) reflection.*
>
> *[Malenfant et al. 1996]*

The authors go on to specify that the word *integral* is key to the concept of reflection because no limits should be placed on what the program may observe or modify.

Maes had also introduced the notion of two levels necessary for reflective systems. Although somewhat informally, they are the self-representation and the internal structures of the program. Here is an extract from the article:

> *A reflective system is a system which incorporates structures representing (aspects of) itself. We call the sum of these structures the self-representation of the system. This self-representation makes it possible for the system to answer questions about itself and support actions on itself. Because the self-representation is causally-connected to the aspects of the system it represents, we can say that:*
>
> *[Maes 1987]*

The need for causal links between the representation of a system and its internal structure becomes apparent. The internal structures should be linked to the representation in such a way that should one change, the other does also. This causality, between what is more commonly known as the *base-level* and the *meta-level* [Genesereth 1983], is necessary to avoid drift between them, which would make reflection very complicated.

In general, the base-level deals with computation about the domain of application, whereas the levels above it, known as meta-levels, perform computations about the system itself. There is no limit to the number of meta-levels, but generally one or two seem appropriate at any given moment in order to avoid complexity. Each meta-level is concerned with the representation and manipulation of the level below it (*i.e.*, its relative base-level), giving rise to the notion of a reflective tower of meta-levels [F. M. Costa *et al.* 2006]. The meta-levels can be seen in Figure 9.

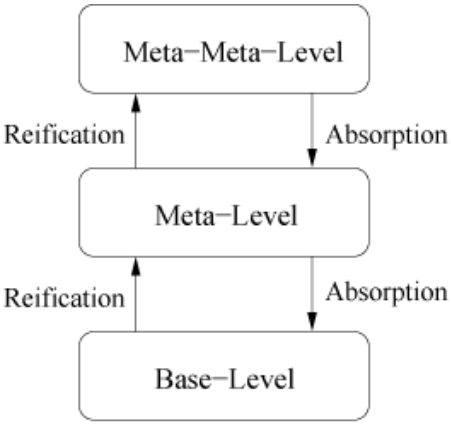

Figure 9: Architecture of a reflective meta-level system.

As shown in the figure, exposing the internals of a base-level to a meta-level is known as *reification*. Modifications to the self-representation result in causal changes to the reified elements of the base-level, which is known as reflection or absorption.

The meta-level provides two types of operations: introspection and intercession. *Introspection* provides the operations of a program to examine the data and instructions of the base-level. It is the ability of a program to reason about itself. *Intercession* comprises the operations which change the data structures and instructions of the base-level. It is the ability of a program to modify its execution.

4.7 Dynamic evolution in software architectures

Software architectures [D. Perry and Wolf 1992] cannot be monolithic. Architectures, just like the software itself, need to evolve in order to satisfy the ever-changing constraints and demands that are placed on them. Software that remains stagnant will be increasingly seen as if it was actually getting worse, and will eventually fall into disuse, as described by Lehman [M.M. Lehman 1980]. Furthermore, software architecture brings modularity to software (*e.g.*, componentization) in order to mitigate complexity. And modularity drives the system to be dynamic [Barais *et al.* 2008]. There are many reasons why dynamic evolution is desirable (*e.g.*, improve cost, ensure safety). The primary notion is to be capable of changing a running system while avoiding interrupting the provision of its services.

To achieve dynamism, an architecture needs to change its configuration, *i.e.*, add new components, remove old components, change bindings and change properties at runtime. This dynamism interacts in subtle ways with the running system and its open computations. The impact of change on running systems needs to be taken into account.

Luckham and Vera [Luckham and Vera 1995], define dynamism in software architectures as:

> *[…] the capability of modeling architectures in which the number of components, connectors, and bindings may vary when the software system is executed.*

Medvidovic and Taylor [Nenad Medvidovic and Richard N. Taylor 1997] describe dynamism as an aspect of component configurations:

> *Configurations exhibit dynamism by allowing replication, insertion, removal, and reconnection of architectural elements or sub-architectures.*

Both of these definitions clearly see an architecture as a composition of architectural elements (*e.g.*, components) that are added, removed and bound to each other. This vision draws the user to a coarse-grain view of the system.

More recently, Baresi provided us with the following definition for dynamic software architectures:

> *[Dynamic architectures] represent systems that do not simply consist of a fixed, static structure, but can react to certain requirements or events by run-time reconfiguration of its components and connections.*
>
> *[Baresi et al. 2004]*

Many Architecture Description Languages (ADLs) have been created to provide for and specify dynamism in software architectures [Bradbury *et al.* 2004]. These languages fall into two main categories concerning the manner in which they manage dynamism, namely by explicitly describing dynamism or by constraining it.

4.7.1 Managing dynamism in software architecture

Dynamic reconfiguration refers to changing the structure of the architecture at runtime. Various authors from different domains such as self-adaptive systems, mobile systems or autonomic computing have used the term. Dynamic reconfiguration is useful for everything from self-healing and context awareness to adding new functionalities. However, dynamic reconfiguration has almost exclusively been used to refer to changing of instances of components, not types.

> *Structural evolution includes changes in both the bindings (connections) between components and the set of component instances.*
>
> *[Jeff Magee and Jeff Kramer 1996]*

In order to distinguish between types and instances, instead of using the generic term dynamic reconfiguration, Cuesta *et al.* [Cuesta *et al.* 2001] have proposed using the terms *Structural Dynamism* to refer to instance management and *Architectural Dynamism* to refer to managing architectural types. Although these terms have not caught on, it is useful to separate the two cases.**Dynamic instance management.** Focuses on changes involving the creation, addition, removal and destruction of components and their connectors (bindings).

- **Dynamic type management.** Focuses on changing the type of an architectural element and its instances at runtime. This is necessary for supporting unforeseen changes.

It should be mentioned that both instance and type management are complementary. Instance management acts at the configuration level (*i.e.* which defines instances and bindings), whereas type management acts at the type level (*i.e.* which defines the behavior and structure).

Although there is a lack of consensus between the approaches to express dynamism in software architectures, it has been argued that the approaches fall into two broad categories regarding how they manage dynamism [Barais *et al.* 2008]. An Architecture Description Language can either support an explicit specification of the architecture's dynamics, which requires evolutions to be foreseen, or, it can define a frame for dynamism in which dynamism is constrained.

- **Explicit dynamism**. Users imperatively specify all changes to be done to the architecture. These ADLs tend to be similar to imperative programming languages. It is common for these ADLs and their dynamism features to refer to architectural instances, and not types, when changing the architecture. All possible future architectures must be foreseen. Examples of these ADLs include Wright [R. Allen *et al.* 1998] and ArchJava [Aldrich *et al.* 2002].

- **Constrained dynamism**. Contrary to an explicit specification of dynamism where all the potential snapshots of the system configuration must be foreseen, other languages try to confine the potential evolution of the software architecture in what can be called constrained dynamic software architecture evolution. These languages provide notions of logical components (*e.g.*, specification) that can be used to refer to a family of components, which all satisfy the constraints to be included in the architecture. Some of

these languages include the notion of Architecture Type. Examples include SafArchie [Barais and Duchien 2005].

Furthermore, although approaches like Darwin [Jeff Magee and Jeff Kramer 1996] have taken a proactive programmed approach to software evolution (see section 4.4.2), other solutions have considered ad-hoc or reactive approaches to evolution [M. A. Wermelinger 1999; Endler and Wei 1992].

4.8 Conclusions

In their attempts to achieve dynamic software evolution, projects have approached the subject in many different ways, as described in this chapter. Furthermore, there are many issues that must be simultaneously resolved in order to allow software to change at runtime.

To be retained from all of this is that software architecture and component-based software engineering move focus from *programming-in-the-small* to *programming-in-the-large*, directing developers' and architects' attention from low-level implementation details to high-level integration concepts. These concepts must include those related to handling dynamism. However, current approaches lack the level of integration necessary to include dynamism into software at design-time while ensuring that the software behaves as expected at runtime. Indeed, dynamism is a cross-cutting concern that affects software from design to execution, over various levels of abstraction, such as source code, packaging, deployment and runtime.

Among the properties that current approaches lack are:

- **Selectively enabling dynamism.** Dynamism is highly invasive and should be enabled where required, allowing developers and architects to concentrate their efforts where dynamism best assists the application in achieving its goals, while saving resources where dynamism is not required.
- **Understanding the impact of dynamism.** Dynamism should be brought under the control of architects. To achieve this goal, the impact of design decisions regarding dynamism must be well understood, even before the application is executed.
- **Ensuring consistency.** Dynamism should never corrupt the application, even when it occurs unexpectedly. Indeed, the tradeoff between selectively enabling dynamism and minimizing the impact of change means that increasing the availability of components not programmed for dynamism can introduce corruption. Consistency is more important than availability.

These factors lead us to include dynamism into application design, and provide design concepts that allow architects to better understand the impact that dynamism will have given their current design decisions. Furthermore, our approach provides guidelines necessary to build components that are designed, decoupled, programmed and packaged in order to ensure proper dynamic behavior.

Part III:

Robusta

Chapter 5

Robusta: An approach to creating dynamic applications

"Problem solving is hardest when all aspects of the problem must be considered simultaneously."
—Stevens, Myers & Constantine, "Structured design", 1974.

"If I had asked people what they wanted, they would have said 'faster horses'."
—Henry Ford.

Software can no longer rely on the purely desktop-centric assumption that its execution environment is static and always known *a priori*, *i.e.*, at design-time. Current areas of research, such as ubiquitous and pervasive computing, are pushing this conclusion to its extreme. Software must handle, among other things, heterogeneity of the underlying execution platforms and communication protocols, mobility of the execution environments (which induces changes to resource and service availability), and the ability to integrate evolving requirements and new features. Altering the software's architecture at runtime, by changing its components and connections, is a mechanism that shows promise in allowing a system to satisfy these ever-changing requirements. Indeed, software architectures allow reasoning about the levels of dynamism required for these new environments and improves the comprehension of dynamic requirements.

This work focuses on giving the software architect control over the level, the nature and the granularity of dynamism that is required in the application. Of particular interest are components that exhibit dynamic behavior, of which we distinguish two types: detachable and volatile. ***Detachable components*** are such that they may be stopped, removed or updated at any time, while ***volatile component*****s** can simply become abruptly unavailable (*e.g.*, because a physical device is disconnected). Applications composed of detachable and volatile components are called ***dynamic applications***. Dynamic applications need to be highly adaptable and resilient, yet they need to remain consistent and to provide best-effort guarantees regarding their availability. They must be developed with these considerations in mind, which means that the availability or unavailability of the services they require should be integrated into both their design and implementation, and no longer handled in a purely *ad hoc* fashion.

However, dynamism is a crosscutting concern that has a particularly large impact on software development. Furthermore, it breaks many assumptions that developers are generally allowed to make. In dynamic applications, developers have to be particularly aware of possible changes that could otherwise corrupt their software and lead to unpredictable execution[21]. Writing dynamic software is complex and error-prone. Arguably, given the level of complexity and the impact dynamism has on development, software cannot become dynamic without (extensive) modification and dynamism cannot be entirely transparent (although much of it may often be externalized or automated).

Nevertheless, recent Software Engineering and Middleware technologies—more specifically those coming from the Component-Based Software Engineering (CBSE) and Service Oriented Computing (SOC) domains—have shown promise in enabling dynamism. Service-oriented component frameworks provide basic mechanisms to handle dynamism, such as dependency injection, late-binding, service availability notifications, lifecycle and dependency management. These mechanisms allow programmers to manage *dynamic service dependencies*; components are programmed to handle their dependencies becoming invalid and changing at runtime. These platforms assist us in tackling some of the complexity related to programming dynamic components but, in practice, they are insufficient when it comes to developing dynamic applications.

Our work specifically targets multi-threaded, synchronous, centralized[22], dynamic, service-oriented component applications. We focus on the following aspects:

From a dynamic application's point of view:

- Determine the inhibitors, if any, to effectively using dynamism in current Service-Oriented Component Models, both in theory and in practice
- Determine the requirements to design and program dynamic components

From an architectural point of view:

- Understand the roles architects and developers play when constructing dynamic software
- Promote dynamism into the design and management of software architectures, where it can be handled as an architectural-concern instead of in an ad-hoc manner in each component
- Allow dynamism to be selectively enabled where it is required, and protect sensitive zones of the application from the instability generated from dynamism

From a tools and platform point of view:

[21] For example, developers must manage the fact that object references they handle implicitly are no longer stable and may change or become invalid at any given moment.

[22] Centralized applications execute in a single address space, commonly known as a process in most operating systems. They may be multi-threaded, where threads share data and references (*e.g.*, global variables) allowing them to communicate effectively and efficiently.

- Propose a software environment that supports and guides developers and architects to produce dynamic software, and
- Produce a framework that manages the application's dynamic behavior during execution.

Interestingly, we found that service-oriented component technologies enforce a level of decoupling that we call ***static decoupling,*** which allows late binding (*i.e.*, two components to be connected at runtime), but unfortunately is insufficient to ensure consistency if the components are disconnected or reconnected at runtime (*i.e.*, if the binding changes to another component or to a new version of the same component). This confusion often misleads programmers into thinking their software will exhibit proper dynamic behavior when it does not[23]. In fact, a higher level of decoupling, which we call ***dynamic decoupling*** — decoupling of both *component types* and *component instances* — is required to ensure consistency in dynamic applications.

5.1 Dynamism requirements and dynamic behavior

Dynamic behavior is the behavior a component *is expected* to exhibit at runtime regarding its lifecycle and the availability of its provided services. It is an architectural declaration used to identify dynamic requirements and calculate dynamic behavioral patterns in the architecture. We have identified the following behaviors:

- ***Dynamic components*** can change the type, number or quantity of services they provide or require at runtime (their dependent components should guard against such changes). There are two types of dynamic components: detachable or volatile.
- ***Detachable components*** can be updated, substituted or removed during the application's execution. Detachable components can be progressively passivated allowing its surrounding components to properly stop in order to avoid corruption (*e.g.*, corrupting the current execution threads).
- ***Volatile components*** can fail immediately and abruptly, which can lead to the corruption of the current execution thread(s) if not handled carefully. Volatile components often represent hardware devices or external services that can become disconnected or unexpectedly inaccessible.
- ***Stable components*** are supposed not to exhibit any dynamic behavior that affects its surrounding components[24]. They are used for zones of the application that remain relatively static and under tight control from the architect, like the core or backbone of the software.

[23] Dynamism requires modularity and modularity is encouraged by static decoupling. However, the inverse is not true, modularity does not require dynamism. The confusion between the two concepts leads to the belief that modular solutions are also dynamic, which is untrue. Nevertheless, dynamic solutions must be modular.

[24] Dynamic behavior is allowed as long as existing (client or provider) components are not affected

- Stable components, during their entire execution, must provide and require the same services and never change their own lifecycle (*e.g.*, unexpectedly shutting down is not allowed).

The declared dynamic behavior of a component is not a direct reference to how the component is programmed, it is related to the behavior the architect expects from it at runtime[25]. The same component, if reused, may exhibit different behaviors in different applications. Identifying the dynamic requirements of the different components of the architecture is a design decision that has repercussions on, among other things, the granularity of dynamism the application will exhibit. For example, declaring that a component is stable means that the architect does not expect the component to change at runtime, allowing other components to rely on this expectation of stability. This generally leads to *groups* of tightly coupled components that do not support dynamism because they are implicitly programmed to expect stability[26]. Nevertheless, in an application, if a stable component fails or becomes potentially inconsistent it must be removed. Given that stable components are often found in groups and that other components rely on the expectation of stability, changing a stable component will often lead to having to change the entire group of tightly coupled components around it, a much larger-grain of dynamism than from a single loosely-coupled (*e.g.*, detachable) component. For example, it is not difficult to imagine that changing a component in the core of the software will most likely lead to the entire core being changed. It is indeed in the interest of the architect to ensure that the frontiers of groups of stable components are properly protected from dynamism in order to avoid the propagation of dynamism to undesirable parts of the architecture.

5.2 Resilience to dynamism

At the component-level, dynamism is seen in two ways: (1) a change in the component's lifecycle[27], and (2) a change in the component's dependencies. ***Dynamic resilience*** is the capacity of a component to resist the inconsistencies or corruption caused from changing its dependencies at runtime. Components must be carefully programmed to enforce the level of resilience that is specified by the architect for each of its dependencies. For each dependency we define the following levels of resilience:

a) ***No resilience*** means that changing the dependency at runtime may cause the component to become corrupt. Static decoupling is sufficient for this level of resilience; it allows late binding but does not ensure proper dynamic behavior if re-bindings should occur.

[25] In practice, the dynamic behavior of a component may be either an intrinsic or a contextual property, that is, either directly related to how it is programmed or related to how it is used. For example, a proxy for an external service might always be volatile because it is programmed that way, forcing the architect to accept, override or ignore its volatility.

[26] A component is said to expect stability if it is not explicitly programmed and properly declared to handle dynamism.

[27] Lifecycle changes imply that a component is programmed to properly start and stop when required. Other lifecycle changes, such as passivation, are used to ensure consistency and minimize the impact of dynamism.

b) ***Dynamic-resilience*** means that the component protects its dependency from dynamic changes, which allows the dependency to change and be re-bound to other components. This requires dynamic decoupling to ensure consistency. A dependency has two types of dynamic resilience: detachable or volatile.

   a. ***Detachable-resilience*** means that the component remains consistent and continues to function properly even when the dependency *changes* at runtime (*e.g.*, substituting the service provider).

   b. ***Volatile-resilience*** means that the component remains consistent and continues to function properly even if the dependency *fails* unexpectedly at runtime. Volatile-resilience, in addition to dynamic decoupling, requires isolation barriers sufficient to protect against failure.

Naturally, the level of resilience a dependency requires is directly related to the dynamic behavior of the components that the dependency is connected to in the architecture. Thus, determining the resilience required for a component is an architectural concern that can hardly be fully understood or properly handled locally (*i.e.*, at the component level). For example, a dependency towards a stable component does not require any resilience at all because there is no expectation of dynamism, yet this information is only known in the architecture; it is an architect's decision to declare a component to be stable and to allow such behaviors. A dependency on a volatile component on the other hand requires volatile-resilience for the component to avoid corruption should the volatile component unexpectedly fail at runtime.

Conversely, once the components' dynamic behavior and the resilience of their dependencies has been established in the architecture, we can calculate if any given component is at risk of corruption or inconsistencies caused by dynamism. We define a component as ***contextually-resilient*** if the component is protected against all *expected* dynamism[28] in the application. This implies that dynamism originating from any detachable or volatile component in the architecture will not propagate to the contextually-resilient component, *i.e.*, it never becomes inconsistent or corrupt from expected dynamism. The architect is charged with rendering stable components contextually-resilient to ensure they will in fact remain stable in the face of dynamism, which allows other components to rely on this expectation of stability. Interestingly, a contextually-resilient component is not required to have any resilient dependencies at all if all its dependencies are stable and contextually-resilient themselves. Contextual-resilience means that the component's level of resilience is sufficient to protect the component from dynamism in all paths originating from this component to all dynamic components. Hence, a component is contextually-resilient if it is protected from *all declared* dynamic changes in the architecture that originate from either detachable or volatile components.

Accordingly, it is possible to render every component contextually-resilient in all possible architectures by making every dependency volatile-resilient[29], which is beneficial for highly

[28] Unexpected dynamism is not calculated, which can occur if, for example, a component's dynamic behavior is incorrectly declared, or, in the event of a component's failure, which is similar to the behavior of volatile components.

[29] This is not entirely true. Although not previously mentioned, the services the component provides may be coupled to its consumer components, requiring that provided services, not only the dependencies, be dynamically decoupled too.

reusable components. However, programming volatile-resilient dependencies is technically difficult and little assistance is currently afforded to developers. Making every dependency of every component volatile-resilient requires expertise, time and effort, all of which are potentially wasted if spent on dependencies that do not immediately benefit the application. Selectively choosing the component's dynamic behavior and the levels of resilience of its dependencies allows concentrating development resources where they are most needed and effective.

5.3 Developing dynamic applications

Even in dynamic applications, many components do not have direct dependencies towards dynamic components. Such components are not subject to the dynamism of other components—they do not need detachable or volatile-resilience—and can generally be programmed without further dynamic restrictions. We call components that do not have dynamic programming restrictions ***dynamic-free components***. Making them resilient to dynamism is, as mentioned previously, a waste of time and effort. A potentially worse side effect of making everything dynamic is that it increases the software's complexity, which in turn reduces its quality and maintainability[30]. Because of the difficulty of programming dynamic-resilient dependencies, the role of the architect is to correctly identify and often maximize the number of dynamic-free components; and conversely, to identify and closely confine the areas in the software that are subject to dynamism.

An important task for the architect is to identify as many dynamic-free components as possible because they can be developed with the same traditional tools and methods used for static software, leading to a potentially faster development cycle with more maintainable code and often better performance, albeit sacrificing the benefits of fine-grain dynamism. In addition, an architect needs to understand how dynamism propagates through the architecture, how and where it can be contained, and which components to protect. For that, the architect may rely on ***component zones.***

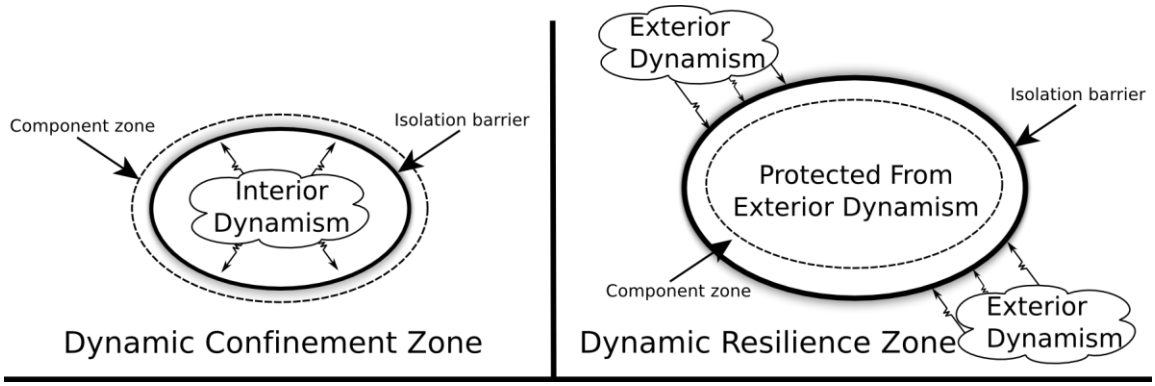

Figure 10: Using zones for the confinement and resilience of dynamism.

[30] Dynamism can affect business logic and thus it also affects the algorithms that an application may use. The result of including dynamism concerns into an algorithm is additional complexity and often a loss in efficiency.

Zones are an architectural construct that represent a connected sub-set of the architecture (*i.e.*, a set of connected components) and, as seen in Figure 10, provide the architect with two important calculations:

1) if dynamic components exist in the zone, verify that dynamism is confined and does not propagate; (we call it a confinement zone) and,
2) if dynamic components exist outside the zone, verify that the zone is protected from exterior dynamism (we call it a resilient zone).

These calculations serve to identify the dependencies that are not sufficiently resilient to protect the zone against *exterior dynamism*, and to identify the dependencies that are not sufficiently resilient to contain dynamism in the zone (and thus affect exterior components). Both calculations serve to answer the questions of *"Can exterior dynamism corrupt the zone?"* and *"Can dynamism escape the zone and affect other components?"*.

Components zones allow abstraction and encapsulation, providing a uniform view of the application at various levels of abstraction. They provide and require services just like atomic components do, and can have their own dynamic behavior (*i.e.*, stable, detachable or volatile). More importantly, for an architect designing a dynamic application, component zones are a fundamental concept because they are an abstraction level and a way to define the structure of the application from the point of view of dynamism. As shown in Figure 11, a zone is composed of a frontier and interior components. ***Frontier components*** have dependencies with, or provide services to, the exterior, while ***interior components*** do not interact in any manner with the exterior. For a zone to confine and be resilient from dynamism, it is necessary for the frontier components to make their services and dependencies resilient to dynamism.

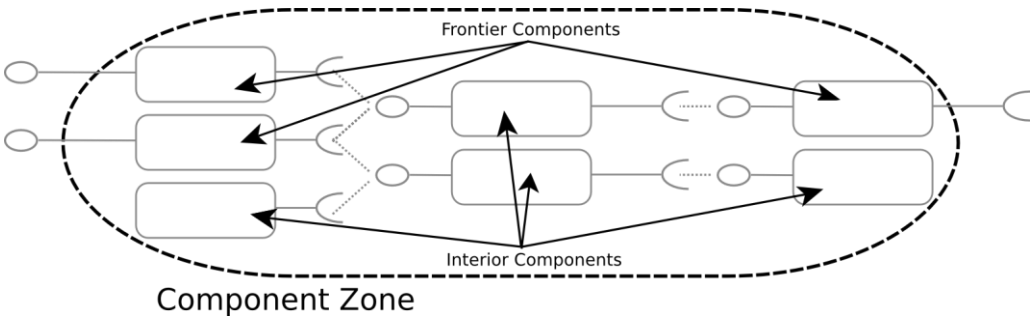

Figure 11: Component zone showing interior and frontier components.

A component zone that is both a confinement and resilient zone is impervious to any expected dynamism. We call such zones, ***dynamic-proof zones.*** A dynamic-proof zone ensures that dynamism that occurs inside the zone does not propagate to any exterior components, and conversely, dynamism from the exterior components does not propagate to the components inside. Additionally, a resilient zone that does not contain any dynamic components is both ***dynamic-proof*** and ***dynamic-free.*** As mentioned earlier, dynamic-free zones allow their interior components (not frontier components though) to be easily programmed without any concern for dynamism.

From the architect's perspective, declaring a zone to be dynamic-proof or dynamic-free becomes an architectural requirement. The system checks if this is indeed true, and if not, it points

out possible dependencies to make resilient. These calculations allow the architect to change his mind, to change the zone's extent or to ask programmers to increase the resilience of the insufficiently guarded dependencies. Component zones are a flexible architectural construct that can be used in different ways by architects. Zones also serve as an architectural aid to identify dynamic behavior, allowing the architect to either confine or protect against dynamism, and improve the understanding and management of dynamism in the application. In short, it is possible to calculate the dynamic behavior of a zone or to verify the behavior that an architect requires. Component zones are presented in more detail in Chapter 8.

5.4 Managing dynamic applications at runtime

A runtime for the execution of dynamic applications is charged with many tasks, such as the deployment and instantiation of components, their removal and destruction, managing the components' lifecycles and their dependencies, minimizing the impact of dynamism on the application, and ensuring proper and consistent execution. Furthermore, the interest of specifying the dynamic behavior of an application and its components is to see such a behavior properly reflected at runtime. Evidently, the design and execution of dynamic applications are intertwined. Our approach ensures that, with the exception of *unexpected dynamism*, the execution of dynamic applications conforms to their design. Given that, at design-time, all expected dynamism has been identified (using the *stable*, *detachable* and *volatile* dynamic behavior declarations) and the safety and possible corruption of components has been pre-calculated, the runtime should not result in any surprising or unexpected dynamic behavior, except what is caused by bugs or by previously unidentified origins of dynamism. Indeed, the calculations serve the architect to understand and manage potential dynamism and its propagation before execution. By *unexpected dynamism*, we refer to software or architectural bugs, such as incorrectly declaring the dynamic behavior of a component (*e.g.*, declaring stable instead of volatile), incorrectly declaring and implementing the required resilience of dependencies, components that crash or return erroneous values, and forcibly replacing stable components at runtime[31]. In such cases, the runtime's priority is to ensure proper execution, which can conflict with declared dynamic behavior leading to otherwise stable components being corrupted and removed. Unexpected dynamism leads to new calculations of corruption to ensure consistency; given any dynamic event that changes the architecture, the runtime must assess if any components have been potentially corrupted and what to do with them. Indeed, resilient dependencies are required to ensure that dynamism at runtime does not corrupt components that the architect has taken the necessary steps to identify and protect[32].

As indicated before, one of the most important tasks the runtime must manage is the preservation of consistency. In order to preserve consistency and avoid the corruption of components, we use an approach that is similar to micro-reboots[Candea et al. 2004]; components that are potentially corrupt are removed and, when possible, others instantiated in their place, leading to a de-facto state of consistency. This is similar in concept to restarting an application

[31] Although the runtime has not calculated and does not expect a stable component to be changed, to the architect or administrator the forcible substitution of a stable component may or may not be a software bug. The architect may simply choose to declare components as stable and then let the runtime ensure consistency if the component is changed.

[32] For various reasons some components may be identified as expendable and intentionally be left unprotected from dynamism.

when a user notices odd behavior; a restart clears the application's state, renews variables and internal references, and allows the application to start from a known, consistent state. Thanks to the additional information our runtime has at hand, such as the dynamic resilience of dependencies, our approach automates such a restart procedure by dynamically computing the minimum number of components that need to be re-started, instead of restarting the entire application. Our method calculates the extent of corruption caused from dynamism and provides for localized recovery.

For every dynamic change, the framework performs a consistency check. Starting from the point of change (*e.g.*, the component that stopped or disappeared), all connecting components are analyzed to see if they are properly protected against the type of dynamism (*e.g.*, detachable components are removed passively while volatile components may stop providing services abruptly). If protected they can continue to be used, but if they are not, they are part of the ***corrupted area,*** which is the list of components that have become potentially corrupt because of dynamism. This means that the dependencies were not resilient against the change that occurred. Once a component is added to the corrupted area, which is the set of components to be removed, all components connecting to it must also be checked for consistency, potentially propagating the corrupted area across the application until sufficiently resilient dependencies are found. Dynamic-free zones are particularly sensitive to corruption, any component in the dynamic-free zone can potentially corrupt the entire zone (because they are tightly coupled and decidedly not resilient), and as such, all components in the zone might be added to the corrupted area. The risk of contamination and corruption are the main reasons architects should protect dynamic-free zones from dynamism. Once the entire corrupted area is identified, it can be removed and necessary components can be replaced[33], bringing the application back into a consistent state. Of course, any non-persistent state in the components themselves may have been lost during the process[34].

Indeed, the runtime's mission is to enable dynamism while avoiding inconsistencies and the potential corruption of components. Furthermore, it needs to minimize the impact of a reconfiguration to the minimal number of components possible. Although there is a general tradeoff between consistency and availability, preserving consistency is generally preferable because it avoids unexpected and non-deterministic behavior caused by potential corruption that originates from improperly handled dynamism. We feel that two of the main reasons that current software does not use existing dynamic approaches are because of the potential for dynamism to silently corrupt the application, and from the lack of tooling to assist developers and architects in statically computing corruption's ripple effect. Furthermore, and although the primary focus of this work is not on software failure, there are parallels between preserving consistency in dynamic applications and preserving consistency in the face of component failures. The mechanisms we propose are effective for the construction of failure-tolerant software as long as the failures are identifiable by the runtime and the executions are interceptable and recoverable. Mechanically, the abrupt failure of a component is treated identically to the abrupt unavailability of a volatile

[33] Our work does not focus on finding, installing and instantiating components necessary for correctly substituting failed services. We delegate these operations to the APAM framework, which has its own logic and procedure regarding selection and dynamic substitution.

[34] Dynamism, much to the same measure as scalability, affects architectural decisions regarding the handling of persistent or "valued" state. State contained inside dynamic components is temporary at best. Permanent state should be offloaded to, for example, persistent back-ends.

component. In such cases, and unlike volatile components, the notion of a *"failure-able component"* is unlikely to be identified at the architectural level (*i.e.*, any component can fail and we do not expect the architect to tell us which ones might fail). This means that design-time calculations are less useful and, for example, the unexpected failure of a stable component may lead to very large parts of the application becoming corrupt. We explore the notion of failure more in section 7.1. However, even though a potentially large number of software failures may be managed using our mechanisms, our work focuses on dynamism.

In short, we propose that architects design applications that either confine or resist dynamism, while the runtime preserves the application's consistency. Design-time calculations enable the architect to identify, localize, and confine the risks that originate from dynamism, while the same calculations at runtime identify potentially inconsistent components and remove them. Our approach allows architects and developers to concentrate their efforts, in regards to dynamism, where needed, while enabling them to design dynamic applications that behave as expected and remain consistent at runtime.

5.5 The rest of this document

The remainder of this document goes into the details of our approach, presents our implementation and then concludes our work. We give a short overview of the chapters to follow:

Chapter 6 explains what Dynamic Decoupling is and how it works. We describe how to decouple component implementations such that they can be added, removed, or substituted individually. We also describe the restrictions on decoupling component instances such that they continue to function properly should their dependencies be changed. Our approach focuses on the Service Contract concept and describes the insufficiencies of reducing contracts to a simple Service Interface.

Chapter 7 details how we protect components from failure and dynamism by means of isolation barriers and recovery mechanisms. We also detail how the mechanisms necessary at runtime to ensure consistency. These mechanisms and calculations are shared at design-time in order for architects to understand the expected dynamic behavior their applications will exhibit.

Chapter 8 we provide an overview of our approach, from design to runtime and back. We also describe the types of analysis that can be performed at both the architectural levels, as well at the component implementation levels, to assist architects and developers respectively in their quest to build dynamic applications.

Chapter 9 describes the implementation and validation of our approach, the Robusta framework. Robusta relies directly on the APAM framework for designing, executing, deploying and running dynamic applications, and indirectly relies on the iPOJO component model and OSGi dynamic module platform.

Chapter 10 presents our conclusion and the perspectives of this work.

Chapter 6

Dynamic-Decoupling

"It is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of […] design decisions which are likely to change […]"

—Parnas 1972.

Designing and building dynamic software systems requires splitting the system into units that are individually deployable, installable, instantiable, destroyable, uninstallable, upgradable and substitutable. Our approach is based on a component approach, where dynamism is handled by manipulating the lifecycle and status of individual components. In order for this to be achievable, components must be decoupled. Coupled components inhibit dynamism because they have been programmed in such a fashion as to be unable to be individually manipulated. (Removing a component while another is coupled to it leads to erroneous or unpredictable behavior.) Coupled components mean we have to change larger sections of the software, losing the fine-grained dynamism from changing individual components. Furthermore, we must identify the extent to which other components are coupled. Of course, as mentioned in the previous chapter, decoupling requires programming and design effort that might not be required for every component (there is a tradeoff between effort and dynamism). Nevertheless, components that do require dynamism must be properly decoupled in order to ensure correct operation.

In this section we explain and detail the two levels of coupling we address in this work, namely coupling between component implementations and coupling between component instances. Our approach focuses on the *service* concept and particularly on the *Service Contract* as a means for decoupling components. Decoupled services are flex points in the architecture that allow for dynamically replacing components at runtime. We propose expanding the Service Contract beyond the common notion of Interface that is used in other centralized component platforms. Conceptually, the Service Contract is to components what the interface is to classes (in means of both indirection and decoupling). The Service Contract is an essential element in the design of a dynamic architecture.

Our objective is to decouple components and to establish the Service Contract in a way that allows for all of the following:

- Allows dynamically changing service providers without destroying consumers (and vice versa).

- Allows hiding implementation details of components (not everything is explicit in the service contract).
- Allows using implementation specific classes in the service contract "transparently". This means that both service provider and service consumer components may use implementation specific classes in service interactions without impacting other components.
- Allows multiple providers and multiple consumers to interact using the same service contract, freely and simultaneously.
- Allows service interactions to use complex objects (it is unacceptable to limit interactions to simple primitive objects that only encapsulate data, as is the case in most distributed solutions).

In essence, our objectives revolve around defining the Service Contract in a way that allows components that use the contract (*i.e.*, components that require or provide services conform to the contract) to evolve independently. There are two levels of dynamism among which dynamic changes have an impact, both requiring different techniques and concepts. The first level of dynamic change regards component implementations, where the coupling that occurs among the components is at the class-level. The second-level regards component instances, where coupling occurs because of shared object instances and objects that require special handling (*e.g.*, objects that have a retention policy).

To achieve our objective, we modify the notion of service contract in two ways: a) the service contract includes the service interface and all of its transitively referenced classes (*e.g.*, data transfer objects), which means that the contract is composed of interfaces and classes that define the objects that transit between components during service invocations; and b) the service contract establishes which objects of an interaction can be freely used, shared or kept by a component, and which objects have a retention policy and need special attention.

Our approach to decoupling component implementations is to modularize our application in such a way as to ensure that the Service Contract, when packaged into modules, is fully independent from the modules that compose the component implementations. This gives the Service Contract, from a design point-of-view, an increased priority over component implementations. To achieve decoupling, we rely on analyzing dependency graphs at the class and module levels, and we provide guidelines to packaging classes into modules that ensure sufficient decoupling for independent evolution of component implementations. It is interesting to note that packaging is generally not visible at the code-level, such that developers are not inherently aware of this level of coupling when programming components. This means that packaging is an orthogonal concern to programming. Nevertheless, coupled modules will dramatically limit the levels of dynamism the application may exhibit. In the case of coupled modules, coupling is analyzed (coupling is directional) in order to ensure the minimal number of modules to satisfy consistency are impacted when a dynamic change occurs.

Our approach to decoupling component instances relies on the notion of shared objects that transit through the component instances' provided and required services. Indeed, we provide a mechanism for defining which objects are managed by a component, and thus, coupled to the

component's lifecycle, and which objects are freely shareable (*i.e.*, decoupled from the component). These concepts are important because, in centralized applications, components may rely on shared memory to, for example, improve performance or reduce resource consumption. In the case of managed objects, the component holding such objects is notified that it must release the object; otherwise, the component is destroyed in order to force the release. Contrary to implementation decoupling, instance decoupling is expressed directly in the Service Interface, making it a programming concern that must be managed at the source code-level.

In the following sections we explain in more detail the concepts of component implementation and component instance decoupling, which we call *dynamic decoupling*.

6.1 Decoupling component implementations

Dynamism requires that components be installed and removed at runtime. The main objective of decoupling implementations is to achieve the individual installation and removal of component implementations without affecting other implementations at runtime. Figure 12 shows a simple architecture, composed of 2 components, showing the possibility of either removing the consumer or the provider component. The goal is to guarantee that the other component will continue to function properly after such a dynamic change.

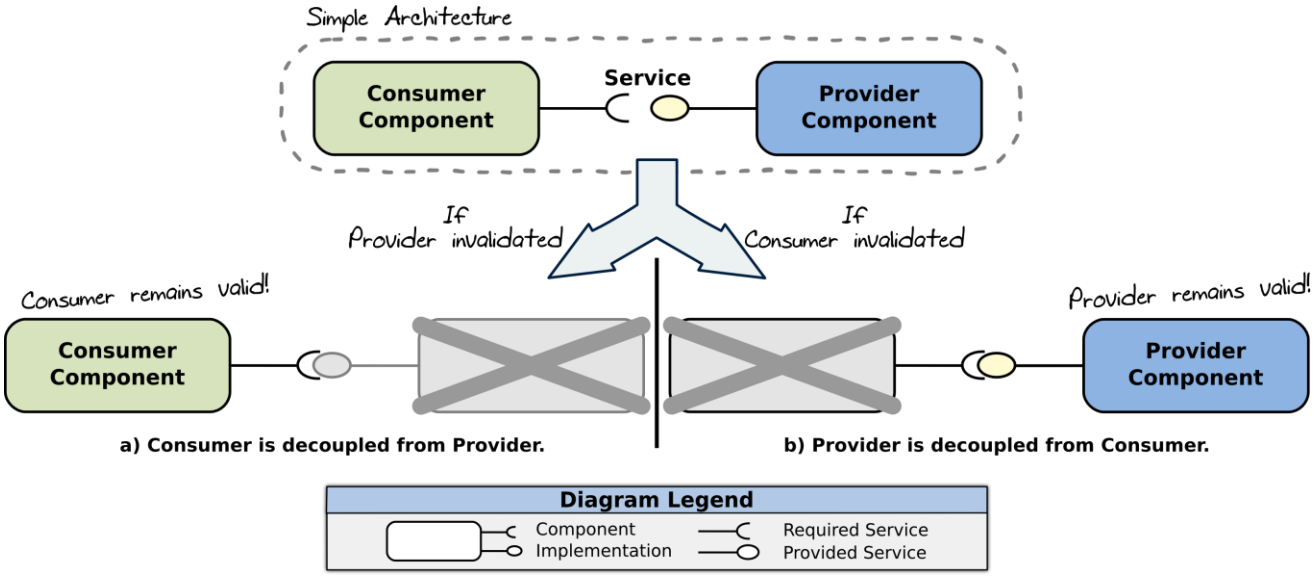

Figure 12: Removing decoupled component implementations does not invalidate other implementations.

Not shown in this figure is that, removing a component implementation invalidates all component instances of that implementation (the instances of a removed implementation must be destroyed) causing dependent component instances to have to rebind to new service providers if available. However, decoupled implementations do not invalidate other component implementations. If the implementations are not properly decoupled then they cannot be individually removed, leading to the invalidation of potentially many component implementations and, consequently, all of their respective component instances. The relationship between component implementations and component instances is similar to that of class and object: if a

class is removed from the platform then all of the objects instantiated from the class must also be removed in order for the class to be properly released. Indeed, the goal of decoupling implementations is to be capable of removing a component implementation with minimal impact on both other component implementations and component instances.

It should be noted that client components may use, either simultaneously or consecutively, multiple service providers, and inversely, service providers may be used by multiple client components. This relationship of N consumers to M providers[35], shown in Figure 13, is fundamental to allowing architectural flexibility. For example, should a provider become unavailable, a different provider that provides the same service can then replace it. Consumers that use different providers should be decoupled from their providers in the sense that, should the provider be removed or become unavailable, the consumer continues to function properly.

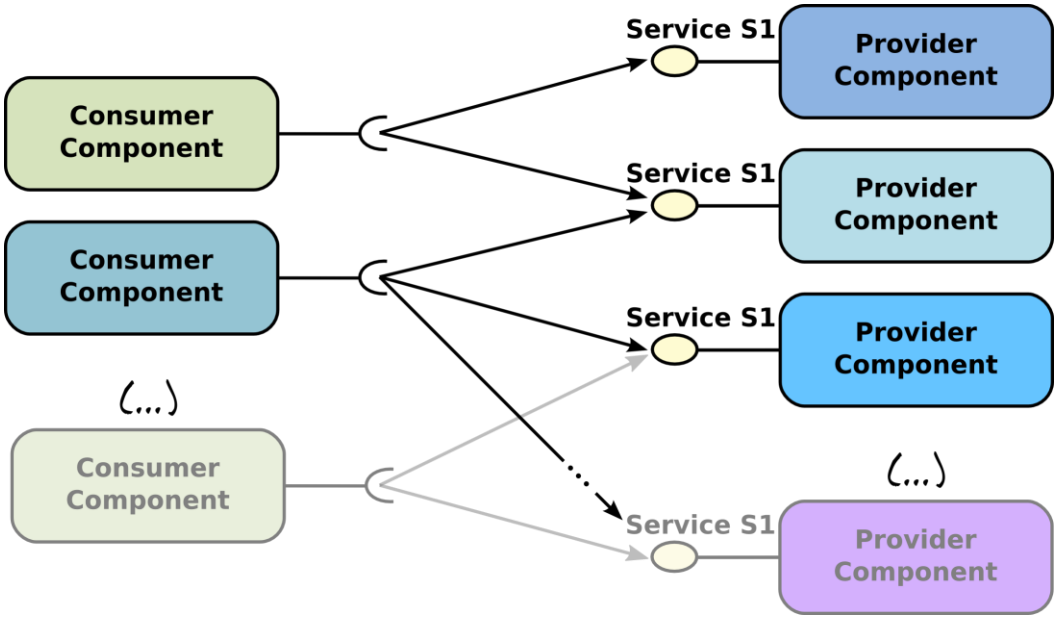

Figure 13: Multiple services defined by a single Service Contract are used by interconnected components simultaneously. Each provider provides the same service and each consumer requires the same service. Decoupling must allow each component implementation to be installed, updated and removed individually, while sharing the same service contract.

Our approach exploits the underlying modularity of the system in order to allow implementations to be individually changed. Modularity refers to how the different implementations are packaged into single deployable units, named modules. This is important because, although components are composed of individual classes that reference each other, a class is not the unit of deployment, modules are[36] (see section 2.5.2). Modules contain multiple classes,

[35] Not all services are required to service more than one client, nor are client services required to use multiple providers simultaneously. However, the N × M relationship is the more general and most demanding case in regards to implementation-decoupling and allowing multiple components to coexist.

[36] There has been much discussion on the relationship between the concepts of class, module, package and component. For our purposes, the Java language groups classes into packages of highly coupled classes (i.e., the Java package). APAM on the other hand, groups packages into larger deployment units that are called modules. There is an informal consensus that a module is a better vehicle for implementations because a Java package is too small. Almost invariably developers arrive at a group of tightly connected packages. Once a cycle exists between two packages, it makes little sense to keep them separate.

so in a sense, we are adding and removing groups of classes in single blocks. Modules are expected to follow best-practices regarding both cohesion and coupling among their classes (among other Object Oriented best practices), but this subject is not treated in our work[37]. Nevertheless, our approach presents further restrictions on packaging which are necessary to ensure proper behavior of dynamic modules. Such restrictions are provided as guidelines, where, if not properly followed, coupling among the modules is detected and fine-grain dynamism (*i.e.*, individual removal of modules) is lost.

This section describes coupling among modules, characteristics required for decoupled implementations and the analyses to detect coupling.

6.1.1 Defining the Service Contract

As previously mentioned, packaging classes into modules is a design decision that affects the dynamic module system. Packaging can cause undesirable coupling among component implementations that leads to the impossibility of individually removing modules (and, by consequence, individual components). In a worst-case scenario, all modules are coupled in such a way that the entire application must be stopped and reloaded for every single change, completely defeating the goal of fine-grain dynamism.

Figure 14 shows a conceptual diagram describing the logical relationship that is commonly perceived between component implementations and the service interface. Indeed, the service contract is generally reduced to simple interface that is shared between two components and is seen simply as the interaction point between them. The implementations are shown as separate entities that do not overlap (in the case of implementations, this means they do not share classes other than the service interface). If we were to modularize such a case, we would probably use three modules, one for the client, one for the provider and a final one for the service interface. Both implementation modules would depend on the interface module, but not on each other (*i.e.*, they are expected to be decoupled).

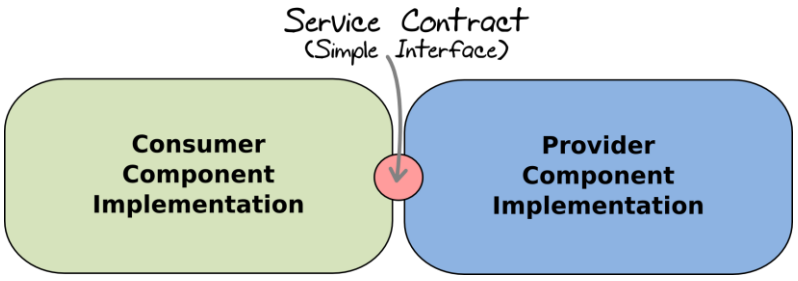

Figure 14: Shows a common misconception of the relationship between component implementations and the service contract in centralized applications (compare to Figure 22).

However, such a relationship is idealistic and does not fit how modularity or services really function in centralized applications. In fact, the service interface is but the *initial* communication point between the components; communication is not limited to the service interface, it can occur through other objects as well. Indeed, the interface defines the operations a provider component

[37] For example, it is possible to package unrelated classes into a single module but this does not make very much sense. We expect modules to contain related classes that have been developed and tested together.

permits; yet, a service interface also references many other classes and interfaces (types) with which both components may continue to interact with. Figure 15 shows a generic metamodel for types (as seen in the Java programming language). A type can be either a class or an interface. Types can reference other types. Also, interfaces can be implemented by various classes and classes can be subsequently inherited (*i.e.*, extended in Java programming language). This shows that a single reference to an interface, *e.g.*, the service interface, can lead to a large graph of other types being indirectly referenced.

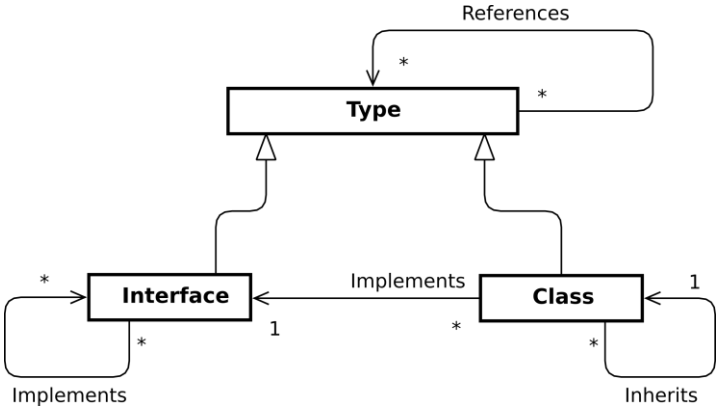

Figure 15: Metamodel showing the relationship between interface and class

Consumer and provider components in a service interaction do not generally reference every type reachable from the service interface. In fact, which exact types are referenced by either component will undoubtedly vary; at a minimal the service interface is referenced by both provider and client, although generally a reasonably large part of the reachable type-graph is referenced by both components. Figure 16 gives us an example of a simple architecture composed of two components that communicate using a single service. The service interface is defined using a Java interface, which is referenced by the consumer component implementation class (necessary for binding and invoking the provider) and is implemented by the provider component implementation class (this is necessary for the consumer to be able to invoke operations on the provider). Note that colors are used to guide the reader into distinguishing between classes that are part of the implementations of each component, and classes that are independent of them and used in interactions (namely the Service Contract classes). Furthermore, it is important to distinguish the component implementation classes from regular classes, and to distinguish the service interface from regular interfaces.

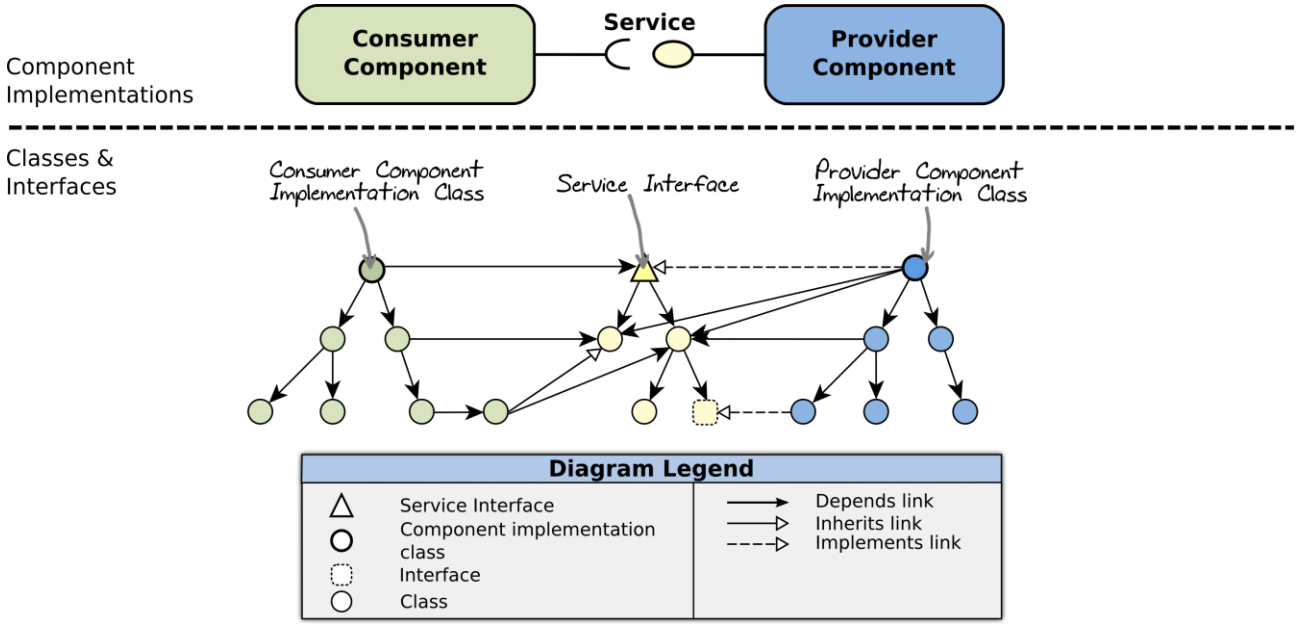

Figure 16: Shows the dependency relationships between classes used to construct components

An interesting result of separating component implementations and the service interface is that we can calculate the entire type-graph (both classes & interfaces) that two components potentially use when communicating through service interactions. This is calculable because objects that traverse component boundaries must be defined by types that are referenced, either directly or indirectly, from the service interface[38]. However, not all types of all the objects that traverse component boundaries can be found in the transitive closure of referenced types starting from the service interface. This is because the transitive closure may contain interfaces or classes that are implemented or inherited by "unknown" types. Indeed, it is possible to know there's an object that passes through the service interface, but we cannot be sure that the object is of the type specified or if it is a subtype (*i.e.*, the service interface can reference a super type to the object that is actually passed in an invocation)[39]. In the case of an interface that is referenced, we know there is a class that implements the interface (otherwise no object would be passed); however, which class is not directly known from the transitive closure[40].

Nevertheless, even though there are still hidden types that can circulate between two components, those types remain inaccessible from the components themselves (components should never downcast objects to find their hidden types because this supposes they know more

[38] It is possible to use either the root class (*e.g.*, Object in Java) or a container object to wrap other objects in ways that the type of the wrapped objects are not known from the type-graph. This implies that the receiver component of the collection casts the objects to a known type, which, in fact, causes hidden coupling between the components and should be avoided. In general, a component should never require down-casting any objects obtained through a service interaction, and, if such is the case, than the service should be re-designed (for example, using generics).

[39] In the worst case, the root class of the class hierarchy is used (*e.g.*, class Object in Java), allowing any object to pass through. This should be avoided when possible.

[40] It's important to note that objects defined by types that are not referenced by a component can still be indirectly referenced and held in memory as long as there is an existing reference towards them. This means that a component programmer may not know he is indirectly referencing a type. This complicates dynamism because if the class is removed, all references, either direct or indirect, to objects of that class must be released, otherwise this will result in a memory leak.

about the objects and the service than what is expressed in the contract, implying hidden coupling). Indeed, a component should only directly reference a type that is contained in the transitive closure of types starting from the service interface, not hidden types.

Understanding the relationship between the service interface and the referenced classes is particularly important because, unlike highly decoupled distributed computing approaches[41], centralized applications can build services using complex objects and interactions. However, such interactions run the risk of introducing hidden coupling, which is difficult to detect and hinders dynamism. Hidden coupling exists in centralized applications because the classes that are used in the service interaction may be unknown and yet still be removed (because of dynamism itself), placing the component programmer in an untenable position where he must release objects that he was unaware of. This affects the components that are directly or indirectly using such classes, without the components (or developers of the components) having any way of knowing which classes are "really" being used. In short, coupled implementations are caused by the (direct or indirect) referencing of classes that belong to another components implementation and which are obtained through service interactions.

Clearly, either component, using or providing a service, can directly reference *any* type (class or interface) that is contained in the *transitive closure* starting from the service interface. As we can begin to see, the service contract must consider the service interface plus additional classes used in the service interaction. This is necessary so we can clearly identify and separate this group of classes from the component implementations. Figure 17 gives us an initial example of which classes require separation (contrast to Figure 16).

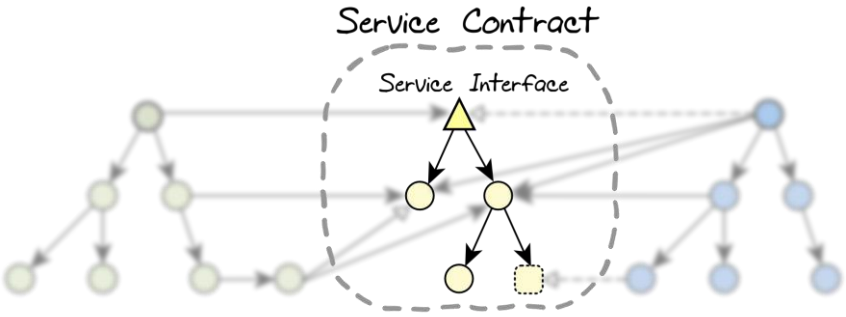

Figure 17: An example showing a graphical view of the Service Contract. The Service Contract is composed of the transitive closure or referenced types reachable from the service interface.

[41]Distributed applications communicate using services with interfaces that define transferable data, generally using primitive data types. Each interaction between consumer and provider passes (*i.e.*, copies) these primitive values between the components. Complex objects, such as classes, are generally not permitted because this causes coupling and reduces the system's flexibility (*e.g.*, this occurs with Java's RMI technology). However, exclusively using primitive types for communication is quite restrictive, undesirable and unnecessary in centralized applications.

Following our explanation, we informally define the Service Contract as follows:

DEFINITION

The ***Service Contract*** of an interface *I* is the set of *types* (interfaces & classes) that are needed to provide the service defined by *I*, *i.e.* all types that *I* directly or transitively depends on.

In order to more formally define the service contract, we will start by defining the relationships between classes and interfaces. The relationship between two types may be caused by any of the following types of coupling found in source code: inheritance, abstract class implementation, interface implementation, composition, aggregation, association, dependency and exception. When considering dynamism as the only concern, we can reduce this to two types of type-coupling, namely the *extends* (and its reverse relationship, *extendedBy*) and the *depends* relationships, which we define as follows:

Definition 6.1 (*Extends relationship*): A type *C extends* a type *D* if its relationship with *D* is of inheritance, abstract class implementation or interface implementation. This implies that *C* depends on *D* at runtime and that *C* provides new functionality not existant in *D*. The *extends* relationship is directional.

Definition 6.2 (*ExtendedBy relationship*): We define *ExtendedBy* as the reverse relationship of *Extends*. *A* type *D* is extended by a type *C* if *C* extends *D*.

Definition 6.3 (*Depends relationship*)**:** A type *C depends* on a type *D* if its relationship with *D* is of composition, aggregation, association, exception, dependency or *extends*. This implies that the type *C* requires *D* at runtime in order to properly function[42]. The *depends* relationship is not symmetric; *C* requires *D* to function but *D* does not require *C* to function.

We proceed by defining the *Type Graph*, which is composed of classes and interfaces:

Definition 6.4 (*Type Graph*): A *type graph* is a tuple $TG = <Type, Extends, ExtendedBy, Depends>$ where:

1. *Type* is a set of types (*i.e.*, classes or interfaces);
2. $Extends \subseteq Type \times Type$ is a partial ordering relation expressing that some types *Extends* (as defined in 6.1) others.
3. $ExtendedBy \subseteq Type \times Type$ is a partial ordering relation expressing that some types are *ExtendedBy* (as defined in 6.2) others. *ExtendedBy* is the reverse relation of *Extends*.
4. $Depends \subseteq Type \times Type$ is a partial ordering relation expressing that some types *Depends* (as defined in 6.3) on others.

We should note that the *Service Contract* is computed by the transitive closure of *Depends* and *Extends* relationships starting from the *service interface* that is used to define the service. For a graph

[42] This is equivalent to a Java import in the class file headers of the Java language. This information is used at compile time to, among other things, ensure type safety, and at runtime for execution purposes. Other languages have similar requirements for both compilation and runtime.

G=<*Node*, *Rel*> with *Node* the set of nodes, and *Rel* the set of relationships between nodes, we note $[r_1, \ldots r_n]^+$ with r_1, $r_n \in Rel$, the transitive closure of relationships r_1 to r_n.

And finally, using the transitive closure operator and the previous definitions, we can define the Service Contract, which is the set of types used to provide a Service.

Definition 6.5 (*Service Contract*): Let *TG* be a Type Graph. A *Service Contract* is a set of types defined by the tuple <*ServiceInterface*, *Depends*, *Extends*>.

1. *SC* (*ServiceInterface*) = { t ∈ *Type* | <*ServiceInterface*, t> ∈ [*Extends* ∪ *Depends*]$^+$ }
2. *ServiceInterface* ∈ *Type* (*ServiceInterface* is the interface used to define the service);
3. Thus, *SC(t)* ⊆ *Type*.

The Service Contract is the set of types reachable, directly or transitively, from the *service interface* trough an *extends* or *depends* relationship.

However, although the Service Contract has been defined, we must still handle hidden objects that can be passed from components through service interactions. In the next section we will detail how we handle such classes while ensuring that components remain consistent and continue to function properly.

6.1.2 Defining the Extended Service Contract

The Service Contract as defined previously describes the types of objects (*i.e.*, the classes and interfaces) that can be passed from one component to another during a service interaction. However, not all classes shared between the components are fully visible using the Service Contract. This happens because the Service Contract can contain interfaces that will be implemented by component implementation classes, and component implementation classes can inherit classes contained in the Service Contract. Indeed, the problem is that of realization and generalization of interfaces and classes. To explain what this means to decoupling implementations, our example starts by ignoring the problem.

Supposing that separating the Service Contract is sufficient to decouple two component implementations, we might believe that creating three modules (one for the consumer component, another for the provider component, and a third for the service contract) would be the minimum number of modules sufficient to ensure proper operation if a component implementation is removed. Figure 18 presents what this would look like if we were to modularize the classes in such a fashion.

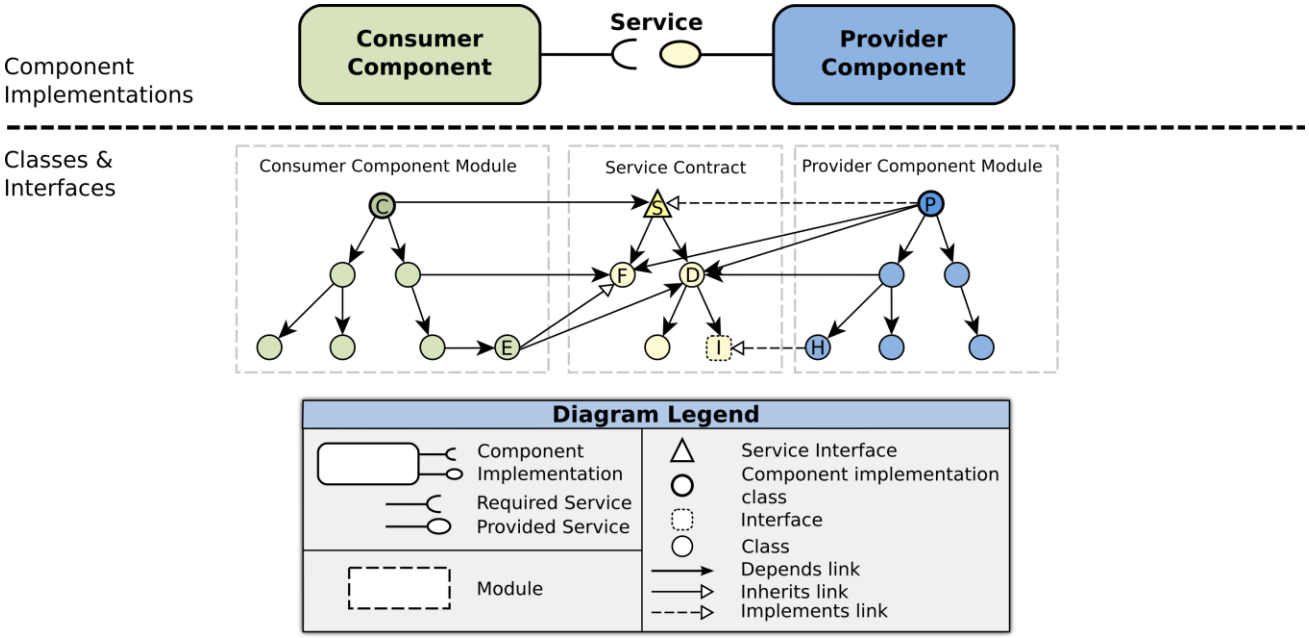

Figure 18: Naïve proposal for packaging a simple component example. Note that class E inherits and class H implements types in the Service Contract. This means that the component implementations may indirectly reference these classes.

As we can see, the component implementations can only directly reference classes in the service contract. There are no direct dependencies between the Consumer and Provider implementation modules. Indeed, such an architecture presents the following desirable characteristics:

- The transitive closure of classes starting from the service interface does not reference component implementation classes. In fact, the service contract is decoupled from the implementation classes because it does not reference any of the component's implementation classes.
- The component implementations reference the service contract.
- The component implementations do not reference each other's implementation classes (*e.g.*, classes of the provider component are unreachable from the consumer component).
- Communication between components is limited to classes that are easily distinguishable and separable (*i.e.*, they reference the Service Contract).

```
1 public interface S {
2     void setF (F f);
3     D getD();
4 }
```

Figure 19: Simple Java interface showing how the Service Interface S directly references classes F and D. Classes F and D are respectively a parameter object sent from the Consumer to the Provider component, and a return value object sent from the Provider to the Consumer component.

However, in Figure 18 we can see that, class E inherits class F which would make it possible for the consumer component to create an instance of class E and pass it to the provider component. (To see how subtle this can be in sour code, we provide the sample interface shown in Figure 19.) The provider component can directly reference class F (*i.e.*, in source code there is a reference to type F); however, indirectly, the provider could be referencing an instance of class E. Should the consumer component implementation be removed, class E would be removed with it, and thus, the provider could potentially hold onto invalid objects of class E[43]. Indeed, this is not the only such case where indirect coupling is visible. Figure 20 presents this and other indirect coupling pathways.

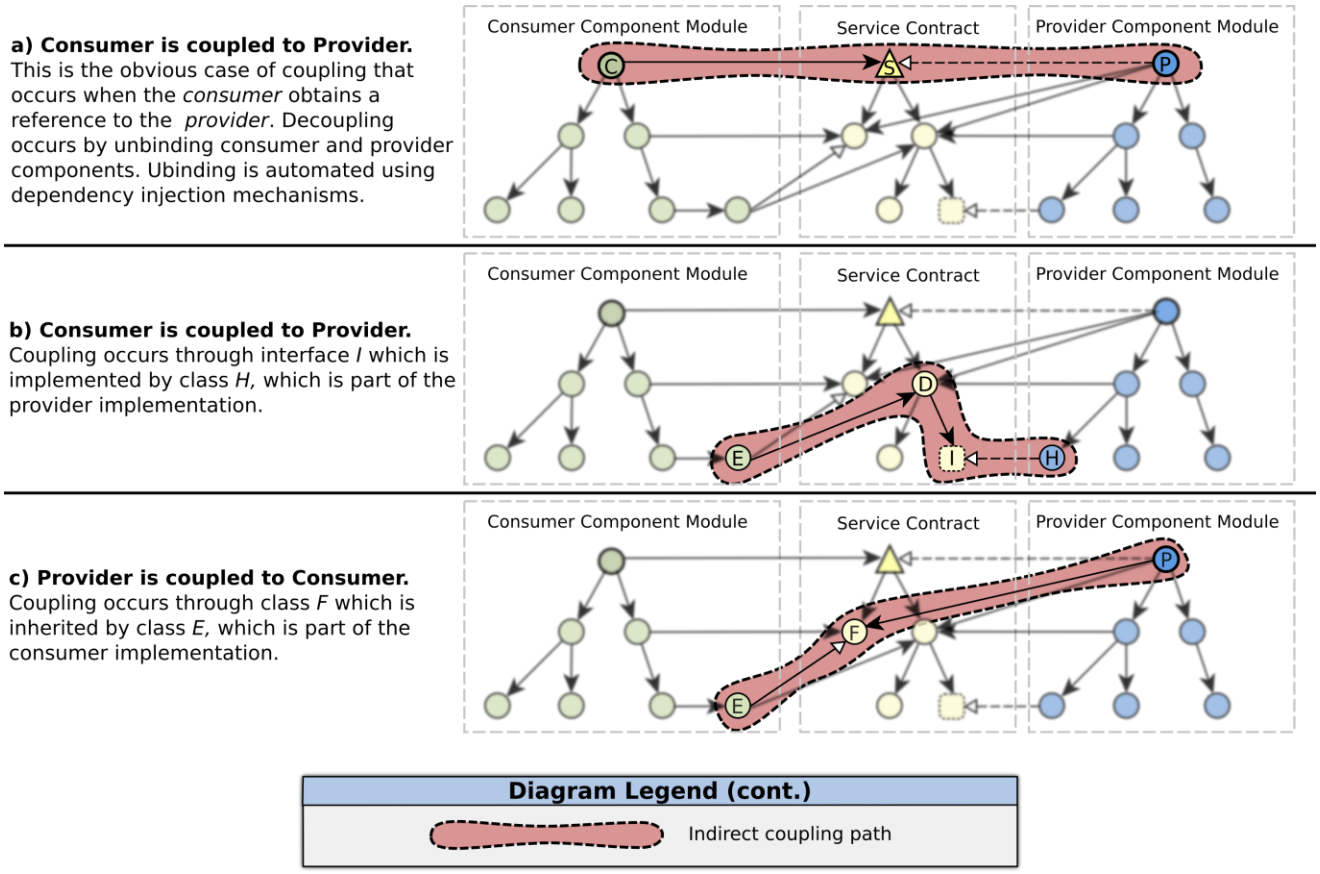

Figure 20: Shows the various indirect coupling paths that occur if we were to follow a naïve modularization technique which does not consider interface realization or class generalizations.

Indeed, indirect references to objects defined by hidden classes will result in memory leaks or unexpected behavior should coupled modules be removed. An initial solution that would work is to outright prohibit the usage of interfaces (other than the service interface) inside of the service contract, and to prohibit inheriting classes from the service contract. This would lead to a solution

[43] Although it is programmatically possible to release the objects of class E that are indirectly referenced, the fact that the component does not know or directly reference class E makes this extremely difficult. This would lead to an overly complicated programming model that is not realistic and would require the programmer to follow all objects of all types he receives through the service in order to release references to those objects in case an indirectly referenced type became invalid. This would be further complicated if there are multiple intermixed subtypes that the component is unaware of, and only some subtypes become invalid.

very similar to distributed systems where all data types that pass among distributed components are explicit, public and shared (and generally composed of primitive data types like integers, strings or floats). Nevertheless, in centralized applications, the drawback of such a solution are that the implementation details of classes, if accessible through the service contract, would need to be made public to other components. Furthermore, components would have to use each and every class exactly as defined in the service contract, disallowing the generalization of services that can be (transparently) specialized. This breaks encapsulation and releases implementation details, leading to an increase in coupling at the source code level (classes will be hard coded to other specialized classes). This would most likely lead to a large number of incompatible yet highly specialized services, caused by potentially small differences or changes in the classes and service interface that build the service contract.

Hidden coupling caused by indirect references is problematic because it is hard to identify (it occurs whenever there is inheritance or implementation of types defined in the service contract), and because the coupled components are not aware of it. In fact, it's invisible to the components themselves; it depends on how they are packaged. In theory, such coupling can be programmatically handled by the components at runtime if the component can release the indirectly referenced objects. For example, in part b) of Figure 20, interface I is implemented by class H, and the consumer component knows, at most, the interface I. Should the provider component's implementation be removed at runtime, the consumer component would have to release all objects defined by class H. Because it does not know class H, the framework could potentially notify the component to release objects defined by classes that implement interface I. However, there could be potentially many classes that implement I, forcing to component to either release all objects or to manage information related to where each I came from and only eliminate the coupled Is.

In general, because the coupled components indirectly reference super types of a coupling class, but not the class itself, it is possible to handle such cases in the source code. If the coupling class is removed, the coupled component can remove references to the objects of that class (*i.e.*, nullify references to the objects so they can be garbage collected). However, because packaging is often expected to be an orthogonal concern[44] we do not expect a component to be programmed to potentially remove all instances of a class it does not directly know, should the framework inform it of such coupling. Certainly, the programming burden would be great because any indirect reference to a class that is removed by the system could potentially invalidate the component and would require complicated programming measures to decouple it at runtime. It is much saner for a component to suppose that any objects it references are held by "stable" classes, allowing a much simpler programming model. Indeed, one of the main benefits of service oriented component models is that you can freely program intra-component (*i.e.*, object-oriented programming paradigm), and you pay special attention to dynamism regarding everything inter-component (i.e., component-oriented programming paradigm). The modularization of software components should follow this logic.

Our approach is to modularize applications with hidden dependencies in a way that the hidden dependencies are held externally from the component's implementation module and held

[44] Packaging becomes an important crosscutting concern that can affect dynamism quite dramatically.

externally from the service contract. This is a compromise between having to make implementation details public and the need to avoid indirect coupling. By putting into separate modules the classes that subtype the service contract, the component's implementation modules can be individually removed without impacting other components, as long as the extension modules and the service contract remain in place. It is important to note that the extension modules can be composed of many types which reference each other, but there should be no references towards the component's implementation module (this would invariably create a link from the service contract to the implementation module because of such a dependency). Continuing with our example, we propose the modules as defined in Figure 21.

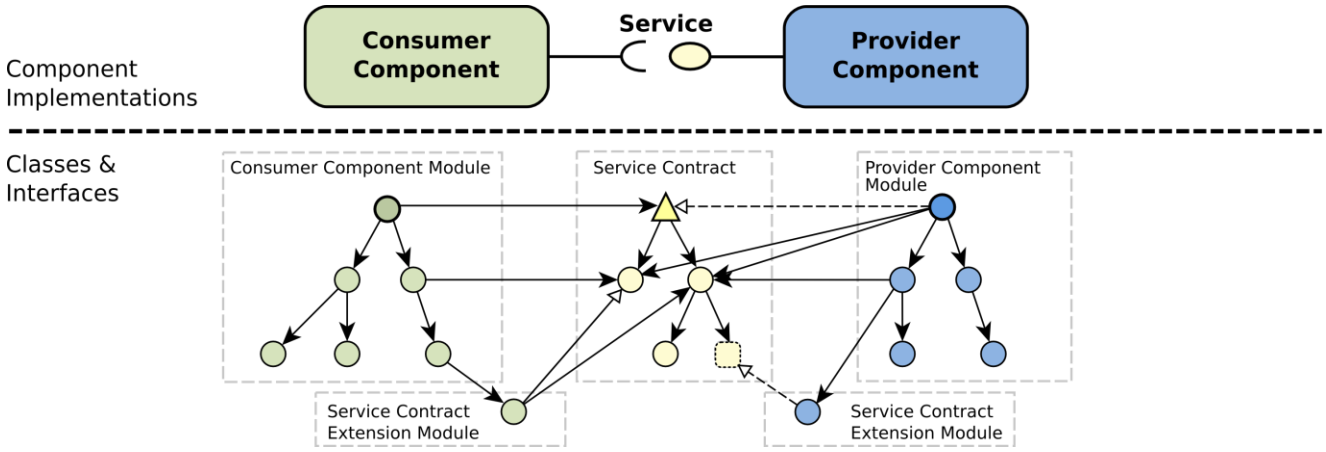

Figure 21: Minimal packaging into modules to allow the component implementations to evolve independently and still allow specializing the service contract.

Indeed, as long as the Service Contract and its Extension Modules do not change, the component implementations may continue to function properly and independently of each other. It should be noted that in Figure 21, we show the minimal number of modules to achieve decoupled implementations. However, although we propose using individual modules for the service contract, for the component implementations and for the extension modules, these could potentially be multiple modules each[45]. Design decisions regarding modularity, such as dividing a module into smaller modules or regrouping modules into larger ones, can be performed to improve reusability, to refactor code, to decouple modules, yet the impact on dynamism should be taken into consideration when making such changes. It is particularly interesting to see that modularity is an orthogonal concern to developing components (it does not directly impact source code)[46]. As a side-note, to achieve greater dynamism we could use one class per module allowing every class of the system to evolve independently. Although it is technically possible to construct a system using one class per module, the runtime cost of doing so would generally be prohibitive[47].

[45] For example, the component implementation may be spread over multiple modules. This does not change our approach. In fact, as long as the set of the modules that is used to construct a component remains decoupled from the set of modules that construct the service contract, our proposition holds.

[46] This is true because at the source code level each class depends on other classes, not modules. The developer is potentially oblivious to where a class will be provided from. This allows modular design decisions to not impact source code.

[47] Dynamic language interpreters, such as Jython (Python on Java), have used similar techniques to allow high-level classes (*e.g.*, Python classes) to be dynamic (*e.g.*, methods and fields can be added and removed at runtime) even though

We see modularity as a crucial concern that needs to be considered when creating dynamic applications. Packaging classes into modules has a large effect on dynamism. Furthermore, the Service Contract is more than just a simple interface as commonly described. As seen in Figure 22 (and contrasted with Figure 14 earlier in this chapter), we show that the Service Contract should not be limited to a simple interface and that it should contain classes that are independent of both provider and consumer implementations. Additionally, it must consider classes that providers and consumers can use in service interactions that serve to extend and specialize the service contract (*e.g.*, classes that inherit from service contract classes or that implement interfaces in the service contract), but which are related to the components' implementations. Extension classes allow components to remain agnostic to how the service is provided (implementation details remain hidden), but to still use the service and keep the objects it has obtained a reference to even though the implementation may be removed.

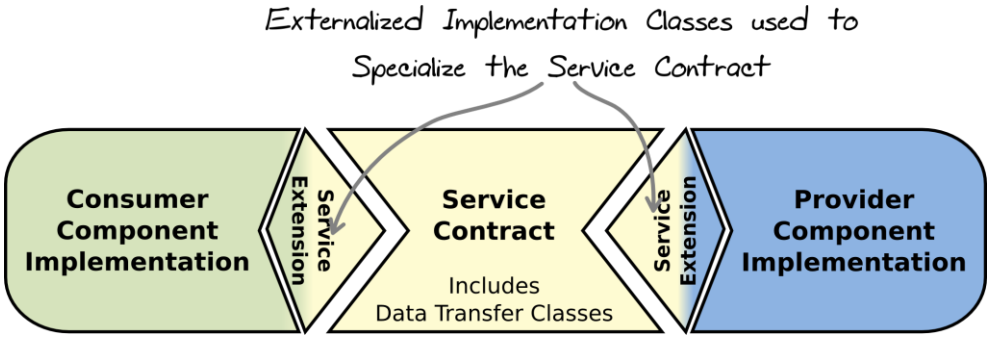

Figure 22: A conceptual overview of the Service Contract showing its importance in the tri-party (consumer, contract, provider) when designing dynamic components (compare to Figure 14).

We informally define the *Extended Service Contract* as follows:

DEFINITION

The ***Extended Service Contract*** of an interface *I* (noted *ESC(I)*) is the set of type that are needed to provide the service defined by *I*, *i.e.* all types that *I* directly or transitively depends on, or indirectly depends on through extensions.

We proceed to formally define the Extended Service Contract using our previous definitions (Definition 6.1-6.5) from section 6.1.1.

Definition 6.6 (*Extended Service Contract)*: Let *TG* be a *Type Graph* ; the *Extended Service Contract* of interface *ServiceInterface*, noted *ESC(ServiceInterface)* is defined by:

1. *ESC(ServiceInterface)* = { t ∈ *Type* | < *ServiceInterface, t*> ∈ [*Extends* ∪ *ExtendedBy* ∪ *Depends*]⁺ } ;

their underlying Java classes are not. To do so, a Jython class is mapped onto multiple Java classes, and each Java class is loaded by its own classloader (very similar to its own module in our approach). If a Python class changes, the Java class implementing that part is removed and a new Java class is loaded (in Java you must remove the entire classloader and all classes it loaded, otherwise nothing is removed, forcing fine-grain dynamism approaches to use one classloader per class). This tends to be very costly at runtime and is probably a reason why dynamic languages on the Java Virtual Machine remain limited in use and are slower than their natively written counterparts.

2. SC(*ServiceInterface*) ⊆ ESC(*ServiceInterface*) ; ESC(*ServiceInterface*) ⊆ Type

The Extended Service Contract is the set of types reachable from the *service interface* trough an *extends, extendedBy* or *depends* relationship. This set of types should be externalized from component implementation modules, into Service Contract and Extension modules, in order for the component implementations to be individually removable.

6.1.3 Modularity: components and modules

In sections 6.1.1 and 6.1.2 we explained the importance of the service contract in centralized applications. Moreover, we have defined the service contract as the set of types reachable from the service interface. We have also recommended that types that inherit or implement the types in the service contract be externalized from the component implementations and put into service extensions, such as to ensure that dangling references do not occur when component implementations are removed. Dangling references are a common occurrence in current approaches given the lack of a *Service Extension* concept, and because the contract between two components is often reduced to a simple service interface[48]. Indeed, the service contract may be specialized using service extensions, and, possibly more commonly, the service extensions may provide implementation classes for interfaces that exist in the service contract.

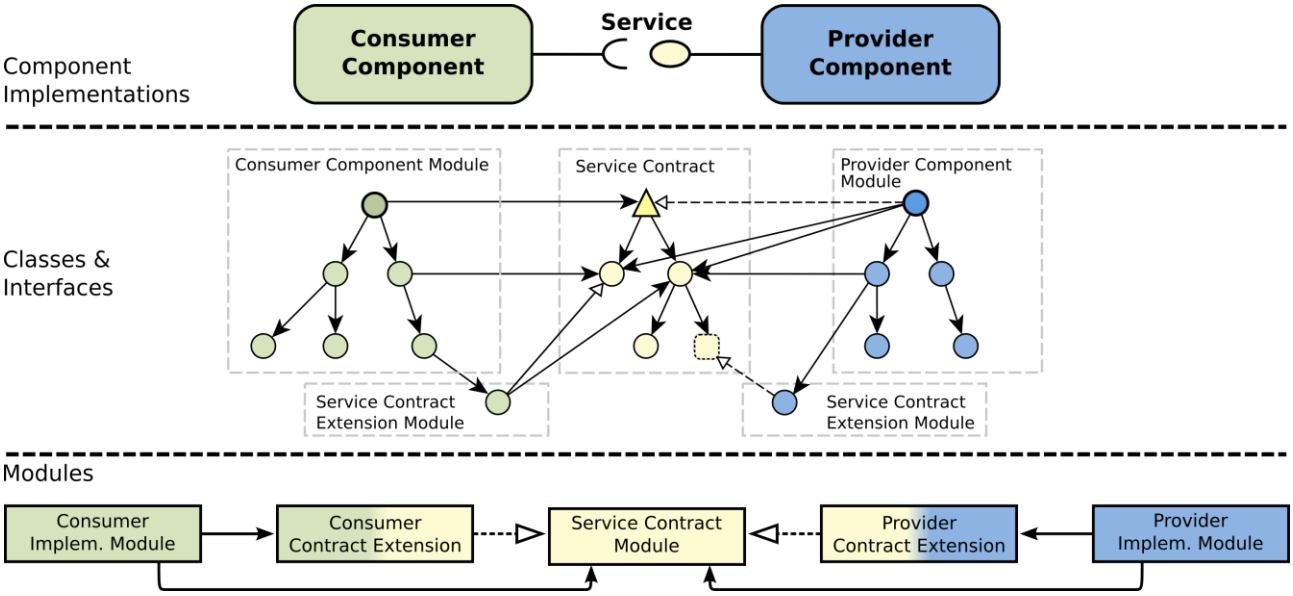

Figure 23: Overview expressing the relationship between components, types and modules

The approach in Robusta is to create five distinct (albeit potentially reusable) modules for every service interaction, such that, either component implementation directly depend on the service contract and their service extension, and indirectly depend on the other component's

[48] Such coupling inhibits dynamism because we cannot safely remove a component implementation because its classes are being used by other components. Doing so may result in memory leaks (*e.g.*, in Java the classloader and the definition of all classes it loaded are transparently held in memory by the virtual machine because of a single dangling reference) or unexpected behavior (*e.g.*, the objects maybe proactively destroyed like in C++). In order to retain overall consistency in such cases of coupling, we have to remove all coupled components, introducing a much larger impact of change.

service extension. For implementations to be decoupled (a requisite for removing and replacing implementations), no direct or indirect dependencies should exist between component implementations[49,50]. Figure 23 reflects the modularization of a simple architecture (consumer component and provider component) with a single service. Visible in the figure (bottom) are the modules that require being created in order for the component implementations to be able to be dynamically removed without impacting other component implementations.

Extension modules depend on the service contract they extend, but they should never depend on the implementation modules[51]. Implementation modules depend on the service contract and service contract extension modules, but not on other implementation modules. The service contract should not directly depend on implementation or extension modules[52]. No cycles between modules should be found in a decoupled modular architecture. Externalizing the service contract and contract extensions is important because, as explained in the previous section, it is possible for component implementations to reference objects defined by classes that are in any of the extension modules (see Figure 20 for a graphical explanation of hidden class-coupling pathways). Indeed, extending a service contract means that the types that are not visible from simply looking at the service contract (*e.g.*, implementation or specialization types) can transit through the service nevertheless. Figure 24 shows the abstract communication channels that exist between two components. We can see that classes that are defined in the service contract and the extension modules are instantiated and "passed" through the service, either as parameters or as return values, depending on the role the component plays in the interaction[53]. Instances of classes contained in the implementation modules do not pass through the service, hence, the component implementation modules are decoupled.

[49] The exception is that indirect coupling always exists from the consumer component's implementation class to the provider component's implementation class. This coupling is a special case that is handled by binding and unbinding component instances, commonly automated using dependency injection and dejection mechanisms. In the case of programmers manually handling (special case) references to service providers, it has been found that it is common for programmers to incorrectly hold references at runtime, leading to dangling service references [Gama and Donsez 2008].

[50] If the implementation of a component A depends, directly or indirectly, on the implementation of component B, it is said that A is coupled to B.

[51] This would create a cyclic dependency causing us to be unable to remove the implementation module, which is one of the main goals of our modular architecture style.

[52] A direct dependency on another module would, by definition, means that the other module is part of the Service Contract. Any module the service contract depends on is actually part of the service contract itself, such that it is impossible to have the Service Contract depend on any other modules. If the service contract and component implementations share modules, they are effectively coupled and cannot be removed separately. Modularizing the service contract is done to ensure that a common base of all reachable types are regrouped into well known decoupled modules in order to ensure dynamism.

[53] The roles may be less clear if, for example, parameters are passed by reference instead of by value, like in C++. Java does not allow passing object references by reference, object references can only be passed by value. However, although it's a bad practice, we can simulate similar behavior by passing an object reference that points to a container object (*e.g.*, a pointer to a hashmap object), which then has object references added to it by the callee. The added references would be accessible to the caller, effectively simulating pass by reference. In either case, complex service invocations do not affect our approach to decoupling implementations.

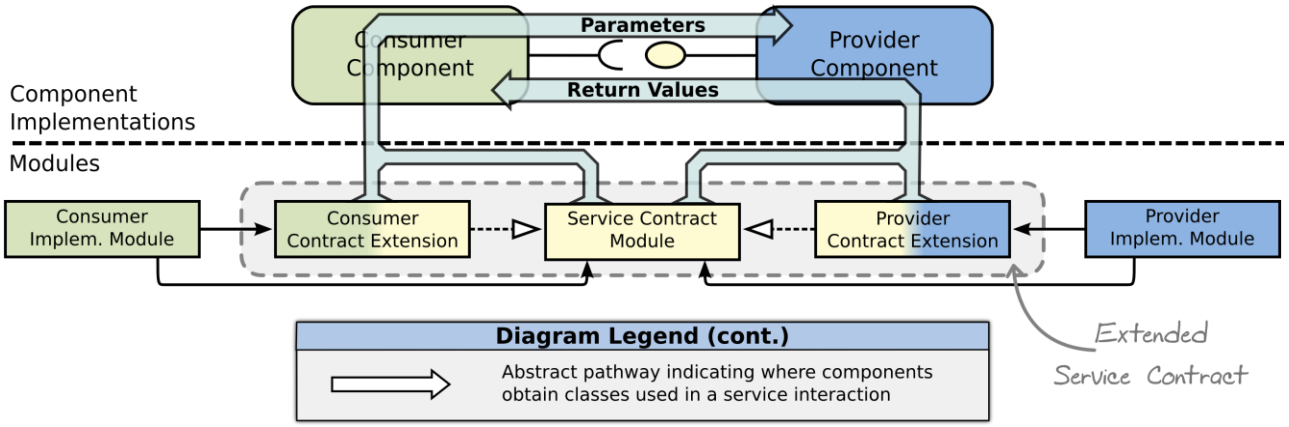

Figure 24: Parameters and return values used in a service interaction are instances of classes that are contained in either the Service Contract module or the Contract Extension Modules.
Components that reference classes contained in a different implementation module are coupled to that implementation.

It is interesting to note that our approach limits types that pass between one component and another to those that exist in the Extended Service Contract. In fact, the service contract is the first delimitation of types that are shared between components (the transitive closure means that only those types and any inherited or implemented types can pass), and is an important reason to why they have been externalized from component implementation modules. However, the Service Contract can still reference the root class in a single root hierarchy language (*e.g.*, the Object class in Java), meaning that any class in the system can pass through the service because all classes inherit from the root class. The design of a service should avoid the use of the root class because this limits most possible analysis to ensure that decoupling is sufficient for dynamism[54]. When encountering the root class in the service contract, if we are optimistic we suppose that the types are properly decoupled, which if wrong could cause dangling references (resulting in a memory leak or other undesirable occurrence); or if we are pessimistic, we suppose that the object can be of any type in the system, resulting in the coupling of the service to all classes accessible by the implementation (which is generally overly strict). In the pessimistic case, it should be understood that any change to any module in the system would potentially require invalidating the service contract and all components that use the contract in order to guarantee there are no dangling references.

In general, implementation decoupling and proper module design allows for the following:

- Component implementation modules can change without impacting other implementations (or instances).
- A single service contract can be used by multiple clients and multiple providers, simultaneously.
- The service contract does not directly depend on implementation or extension modules, it is decoupled and independent.

[54] This problem is similar to that of container classes before the use of generics. The compiler was unable to verify if the insertion or removal (or any other operation) of an object into or from the container would succeed. The risk was that the object was not of the correct type, resulting in an unexpected runtime exception.

- Component instances remain valid as long as the extended service contract is not changed (*i.e.*, the service contract and the contract extensions modules are not removed or invalidated).
- Component implementations may use specialized classes in service interactions; neither the service contract nor the component implementations are required to have previous knowledge of all types that are used in the interaction[55]. The extended service contract continues to allow implementation hiding and encapsulation for centralized applications.

6.1.3.1 Module Dependencies

In Robusta, a module contains classes, interfaces, metadata files, and other file-based resources. Figure 25 shows the file-based metamodel of a module's contents.

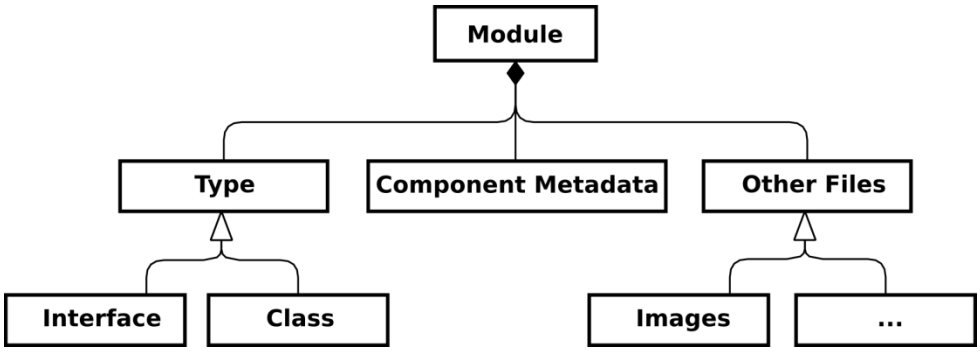

Figure 25: Metamodel showing different files that modules contain

Module dependencies must reflect the relationships between the types (classes and interfaces) contained within the modules. The relationship between two types can be of *Extends*, *ExtendedBy* or *Depends*, as defined in section 6.1.1.

Furthermore, type relationships can be expressed as a directed graph, such as shown in Figure 23 (middle) and formally defined in Definition 6.4. Because types (classes or interfaces) are not individually dynamic—they are grouped into dynamic modules—there is little need to work with such a fine-grained graph[56]. We convert the graph into a module graph that is both simpler (less edges and vertices) and expresses dynamism (modules can be individually installed and removed). To convert the graph we use the following formal definitions:

[55] This is a strong difference with highly decoupled distributed applications, where the service contract is often reduced to simple types, such as Data Transfer Objects (DTO), that are composed of primitive data types like integers, floats, Booleans and strings. Marshaling and unmarshaling of DTOs require both nodes of the system to have a precise representation of the data types. Any change, even potentially simple ones, to a data type in the distributed service impacts both nodes.

[56] Using the fine-grained type graph provides tooling with enough information to propose which classes should go into which modules in order to create decoupled provider and consumer components. See Chapter 8 for more information on tooling to assist in the design of dynamic applications.

Definition 6.7 (*Module Graph*): The *Module Graph* is defined by *MG= <Module, MExtends, MExtendedBy, MDepends>*, where

1. *Module* are the set of modules in the system;
2. *MExtends* $\subseteq$ *Module* $\times$ *Module* is a partial ordering relation expressing that some modules extend others;
3. *MExtendedBy* $\subseteq$ *Module* $\times$ *Module* is a partial ordering relation expressing that some modules are *MExtendedBy* others. *MExtendedBy* is the reverse relation of *MExtends*.
4. *MDepends* $\subseteq$ *Module* $\times$ *Module* is a partial ordering relation expressing that some modules depend on others.

Definition 6.8 (*Type Graph to Module Graph*): Provided a type graph *TG=<Type, Extends, ExtendedBy, Depends>* and a *packaging* function, we can derive the associated module graph *MG=<Module, MExtends, MExtendedBy, MDepends>* as follows:

1. *packaging: Type* $\rightarrow$ *Module* is a mapping function that defines what module a type has been packaged into[57];
   a. $\forall t \in Type \; \exists \; m \in Module \; / \; m = packaging(t)$
2. $\forall t_1, t_2 \in Types \; / \; packaging(t_1) \neq packaging(t_2)$ then
   a. $<t_1,t_2> \in Extends \Longrightarrow <packaging(t_1),packaging(t_2)> \in MExtends$
   b. $<t_1,t_2> \in ExtendedBy \Longrightarrow <packaging(t_1),packaging(t_2)> \in MExtendedBy$
   c. $<t_1,t_2> \in Depends \Longrightarrow <packaging(t_1),packaging(t_2)> \in MDepends$

The following metamodel, shown in Figure 26, graphically shows the relationships between modules obtained from the graph manipulations (we have omitted the *ExtendedBy* relationship because it is the reverse of the *extends* relationship).

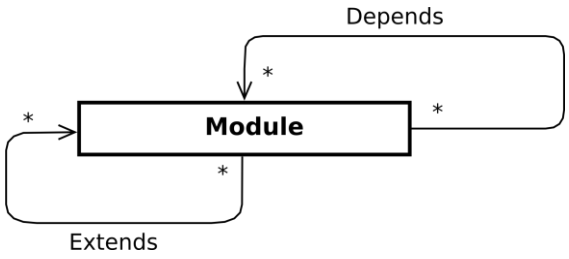

Figure 26: Module Relationship Metamodel

[57] In practice packaging is performed by the developers who decide what modules to create and then select which classes go into those modules.

6.1.4 Service Contract reusability for multiple components

Our approach to decoupling component implementation makes the service contract and contract extensions a central element in the design of dynamic applications. It is important for components that provide and require the same service to use the same service contract modules. This ensures that components are using the same type definitions in order for them to be able to interact without running into problems caused by incompatible or unavailable types. Figure 27 presents an architecture with multiple components (multiple provider and multiple consumer components) interacting through the same service (namely service *S1*). The result of such a scenario means that objects obtained from one provider can intermix with objects obtained from a different provider, and *vice versa* with consumer produced objects. Furthermore, objects created by one provider component can be passed to a consumer component, which are then passed to a different provider component, and then passed to another (entirely different) consumer component. Because every component implementation depends on the same service contract, if the objects are instances of classes in the service contract, this is not problematic; however, if the instances are from classes in the extension modules, this means that the components are indirectly coupled to any or all extension modules. Indeed, the idea of externalizing extension modules is to combat indirect coupling. All components that interact with a service are potentially indirectly coupled to any or all extension modules.

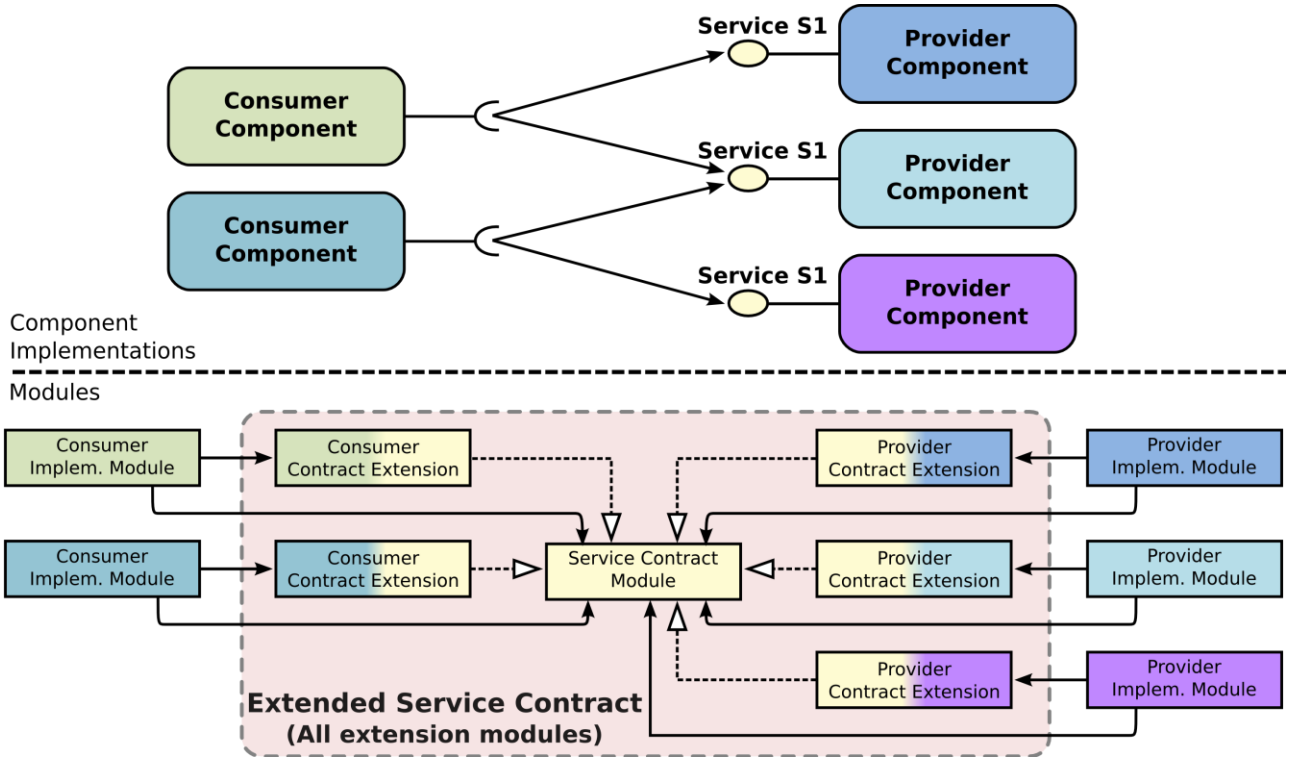

Figure 27: Multiple client components bound to multiple provider components, all around the same service contract. By allowing each individual component implementation to be installed and removed independently, our approach to implementation decoupling solves this otherwise complicated case of coupling among modules. Dynamism is limited to implementation modules, which can be freely installed, removed or updated. However, The Extended Service Contract must not be changed (no modules can be removed, yet more modules can be added) because this could result in dangling references.

This level of dynamism effectively allows implementation modules to be added, removed and updated without ever invalidating other component implementations or the service contract. Service extension modules are less dynamic, you can add new service extensions when, for example, installing new component implementations; however, service extension modules cannot be freely removed without potentially causing dangling references because of indirect coupling. To avoid such dangling references, the framework would have to destroy all instances of all components that use the service. The Service Contract itself cannot be changed in any way; no classes or modules can be dynamically added to it and it cannot be removed without invalidating all modules (and all components) that depend on it.

An illustration of what this means in regards to designers and architects using high-level architectural concepts is provided in Figure 28. As we can see, the service contract is central to all components that require the service or provide the service. For the components to be compatible with one another, they must use the exact same contract (all contract classes must be the same). Adding new components, either consumer or provider, can be done without impacting other components. There may be any number of consumer or provider components, all using the same service. Component implementations may be removed without impacting other components as long as they are properly decoupled and their service extension remains in place. Finally, service extensions do not have to be unique or per-component; two components can reuse the same service extension (effectively reusing the same underlying modules to instantiate the classes needed to provide the service) without any issues. All in all, this allows for a fairly flexible modular solution that enables and encourages dynamism.

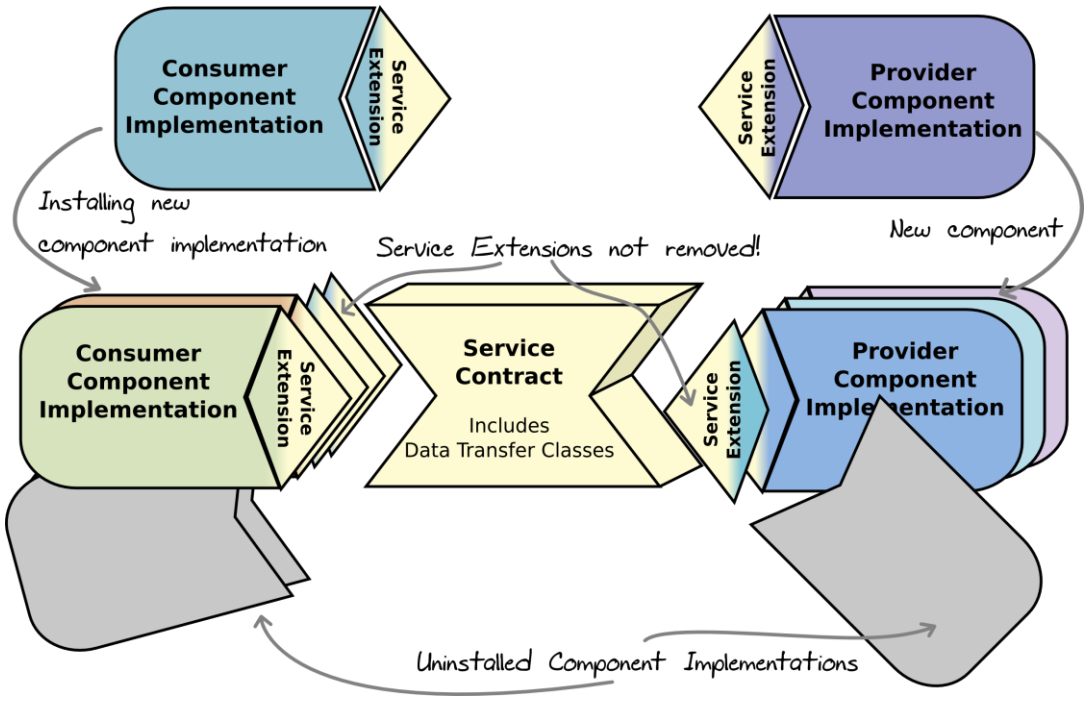

Figure 28: Conceptual representation of multiple component implementations using a single service contract. Given that the Service Contract is decoupled and well identified, all component services are compatible and can interact through this single service. Note that as long as none of the Service Extensions are removed, component instances instantiated from these implementations will remain valid. If an Extension is removed, this can lead to having to invalidate and destroy all component instances in order to avoid dangling references.

6.1.4.1 Adding dynamism to the Extended Service Contract

Up to this point we have expressed the need to ensure that classes used in a service interaction not be removed because this would lead to dangling references, which cause memory leaks and potentially other unexpected behaviors. At the module level, in order to ensure that dynamism is properly handled, we have established that we cannot remove any module from the set of modules in an Extended Service Contract without risk. Interestingly, if we analyze the component architecture and the module architecture at the same time, we can have a better estimation of components that are potentially coupled to service extensions. Figure 29 shows a simple architecture with three components. Using our approach, we can see that the components A, B and C are potentially coupled to any of the service extension modules. Indeed, without thorough dynamic runtime tracing or instrumentation we cannot effectively know if, for example, component B holds a reference to an object from component C's service extension. Of course, analyzing the architecture at the component instance level, at runtime, would give us a more precise calculation.

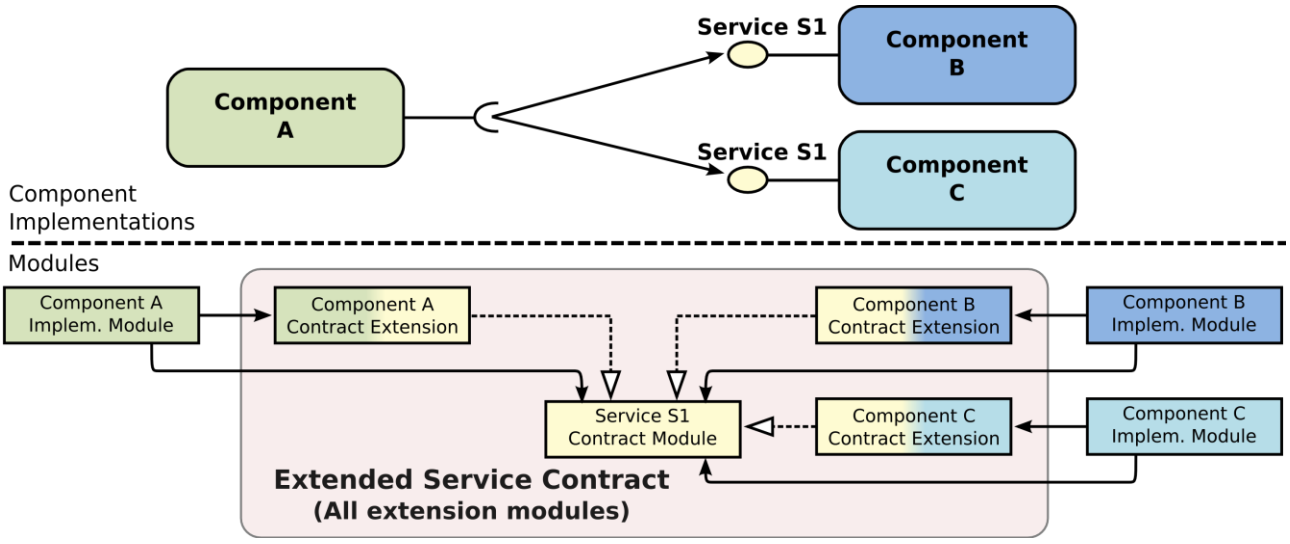

Figure 29: Because of the interaction path through component A, component B is coupled to component C's Service Contract Extension (and vice versa).

Looking at a different architecture composed of two disjoint sub-graphs, as can be seen in Figure 30, we can see that a module-level analysis would imply that Component B is potentially coupled to component C's service extension. However, components B and C are in disjoint graphs at the component implementation level, meaning that there is no possible path for objects from C's service extension to be passed to component B (nor vice versa). If component C's service extension module were to be removed, there is no reason to invalidate component B. Indeed, by analyzing the architecture and possible interaction paths we can improve our analysis of coupling and change impact.

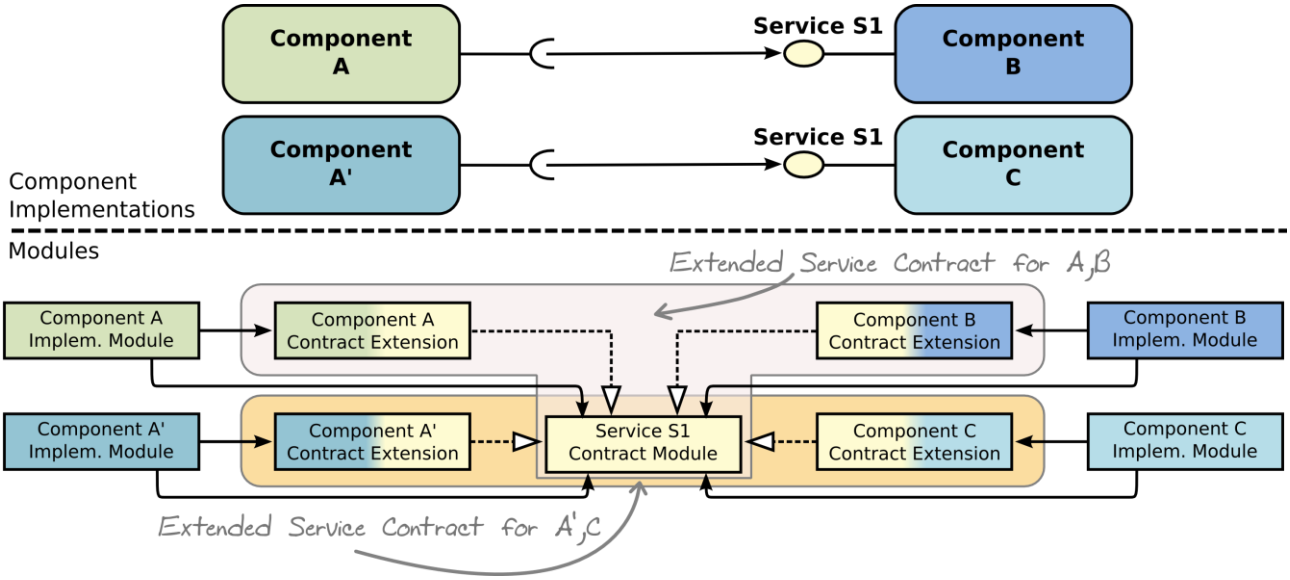

Figure 30: Shows that because there is no interaction path through which to share objects, component B is decoupled from component C's Service Contract Extension (and vice versa).

6.1.4.2 Lazy removal of Service Extensions

Removing a service extension runs the risk of invalidating all components currently using the service in order to be sure no component is directly or indirectly using classes defined in the service extension module. Components directly using a module's classes can be trivially found; those indirectly using them are much more difficult to detect. Although removing a service extension without invalidating indirectly coupled components can lead to dangling references, which, given the execution environment can lead to different runtime problems, it is interesting to note that in controlled circumstances we can remove the service extension lazily. In the particular case of the Java framework, and when no modules directly rely on the service extension module, (*e.g.*, the component implementation that uses the extension is removed), and the extension is properly decoupled, we can remove the extension from the framework without having to invalidate indirectly coupled components. Underlying, the Java framework will only free the classloader and loaded classes from the extension module when all objects created from the classes in the extension module are released. Without thorough dynamic runtime instrumentation, we cannot be sure if all objects have been released or not. Nevertheless, this doesn't really matter because, if they have not been released[58], the memory used for the classloader and classes will not be freed, hence we still consume the same amount of memory as before the extension module was removed; and if the objects have been released, the Java virtual machine will recover the used memory[59]. Furthermore, even if the objects aren't immediately released, if they are eventually released than we will recover the memory at that time.

[58] We call modules that the framework has removed but which the underlying virtual machine has not garbage collected because of indirect or direct references shadow modules. Shadow modules can be very difficult to detect.

[59] It is particularly important to manage the memory consumed by classes and classloaders because in many Java virtual machines, classes are loaded into the permgen memory space, which is a much smaller memory space used to store class definitions, and should not be confused with the heap memory where objects are stored and which is generally much larger. Filling either the heap or the permgen will result in an out of memory exception.

In such cases, it can be in the application's best interest to remove unused service extensions given the possibility that used memory might be recovered. The only immediate drawback of doing so is that the classes in the module, even if it is not removed because of indirect coupling, will not be usable or loadable by other modules should a new component be deployed and require them. In such cases, we would have to deploy a new module or redeploy the same module to satisfy the class dependencies, potentially causing a shadow version of the module and an active version to coexist. This is not particularly dangerous, there is no limitation to having multiple versions of the same classes available in the framework simultaneously; however, it is a waste of memory (the exact contrary of our objective of recovering memory) and should be a technique that is used sparingly. This work has not explored more effective techniques to discover unused modules.

6.2 Decoupling component instances

Component instance decoupling is the second part in our approach to decoupling components in order to ensure dynamism. Dynamism can drastically impact an application, creating a complex programming model that is both difficult to understand and error-prone. In the previous section we analyzed and proposed an approach to modularity that allows component implementations to be added and removed without impacting other component implementations. Nevertheless, and not to be confused with implementation decoupling, there is a type of coupling that is not related to where a type definition is stored (*i.e.*, what module contains the class or interface), it is related to how a component uses the objects it obtains through a service interaction and to additional constraints the object might have. We call this instance decoupling.

Instance decoupling has a direct impact on programming because it expects components (or developers as it happens) to follow guidelines on how long a component can hold a reference to an object. We propose that developers rely on the concepts of freely shareable objects (in a very large sense of shared), and managed shareable objects (also in a very large sense of managed). Both types of objects transit through the component instances' provided and required services through service interactions. The approach is very similar to a data retention plan, where components can specify which objects are managed and which objects are free. Particularly, a reference towards a managed object must be released after a certain event, while free objects can be held indefinitely[60].

We define which objects are managed by a component (*i.e.*, coupled to the component that provides the object) in the Service Interface, as well as which objects are freely shareable (*i.e.*, decoupled from the component). The concepts of free and managed objects are important because, in centralized applications, components may rely on shared memory to, for example, improve performance or reduce resource consumption. Furthermore, some objects require special semantics or are not usable after some events (*e.g.*, configuration objects). The component holding managed objects is notified that it must release the objects; otherwise, the component is destroyed in order to force their release. Contrary to implementation decoupling, instance decoupling is expressed

[60] Note that this is orthogonal to the modularity concern (it does not matter if the component implementations are coupled or not) and it is not transparent to development. Programmers must know if an object is managed or free and must respect the fact that it must be released if managed.

directly in the Service Interface, making it a programming concern that must be managed at the code-level, as shown in Figure 31.

```
public interface PrinterService{
      void Print (@Managed Document D);
      @Free Status getStatus();
      @Managed[61] PrinterConfigurator getConfigurator();
}
```

Figure 31: An example printer service showing the use of Managed and Free object annotations on return values and parameters in the Service Interface.

6.2.1 The need for Free and Managed objects

Developing dynamic applications is uneasy because of all the restrictions that are added to the programming model in order to ensure that the applications will be consistent and continue to function correctly after a dynamic event (*e.g.*, a change event, a failure event). Decoupling components can be pushed to its extremes, leading to approaches similar to those found in modern distributed computing solutions where only simple data structures are shared (composed of primitive data types), or at most, xml documents with well-established document type definitions. This places the burden of decoupling on the definition of the data that circulates between the nodes and prohibits the use of complex objects. If the definitions change, both nodes are impacted, because encapsulation and implementation hiding are no longer possible.

The requirements when developing dynamic applications in centralized environments are much less binary; decoupling components is a tradeoff between encapsulation (and implementation hiding) and increased dynamism. Retention policies on data or objects obtained through a service also require, in some cases, special distinction. When an object is shared through a service, it is often desirable to make explicit the conditions under which it can be safely used, in order to ensure that it will not be invoked when in an undesirable state or held onto beyond a certain period. Although there are various ways in current approaches to simulate such behavior[62], we feel there is still a requirement to establish, at the service contract-level, an object retention policy that makes explicit such a need to ensure that objects are released when required. This allows sharing potentially complex objects, such as database connection objects and thread objects, through simple service invocations, with the assurance that the objects will be released when the component that provided them is removed.

Retention policies can become extremely fine-grained according to the requirements of each application. For example, we can envision a retention policy that is independent of the component's lifecycle (*e.g.*, a complex object that a component can invalidate at any moment), or

[61] Annotations in Java cannot be applied to return values; however, they may be applied to methods. Because a method may have only one return value, we interpret the annotation as being applied to the return value. The compiler isn't generally too picky, so it is possible for readability purposes, as shown in the example, to make it appear as if the return value is annotated.

[62] For example, in the Java programming language you can use the `IllegalStateException` in order to throw an unexpected runtime exception in case a method invocation is performed on the object at an unacceptable time.

one that is coupled to the component's implementation's lifecycle (*e.g.*, useful for static or singleton values, methods, classes that are shared among component instances), but these tend to be side-cases. Therefore, in this work, we limit retention policies to a single possibility: Managed Objects are linked to the component instance that has created them and must be released when the component instance is invalidated or destroyed. For example, if we have a `DatabaseConnectionPool` component, we expect all other components to release `DatabaseConnectionObjects` when the component is invalidated. Indeed, we feel that coupling objects to the component instance's lifecycle covers a large majority of cases that require the use of Managed objects.

6.2.2 Free and Managed objects

Figure 32 shows how object references from consumer and service component instances can point to the same object in memory. When two components point to the same object, independently of the object's class, the object is considered a shared object. Any parameter sent from a consumer component to a provider component can become a shared object. Any return value sent from provider component to consumer component can become a shared object. Furthermore, a shared object can point to other objects, leading to an object graph being shared.

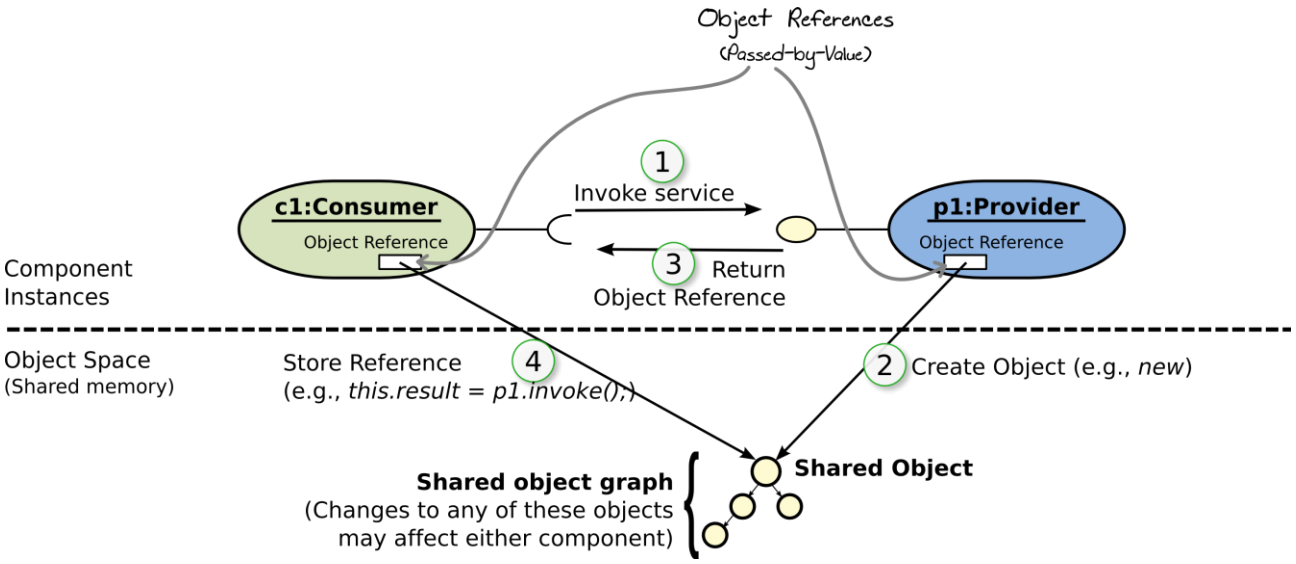

Figure 32: Invoking a service causes parameters to be passed to the service provider and return values to be sent to the service consumer. Object references from either component can then point to the same object, which is considered shared.

Shared objects, often enough, represent a resource used in common by client and provider components (*e.g.*, a port, a configuration file, a thread, a database connection object), or store shared state (*e.g.*, a session). If, for example, the provider component disappears, the shared object is possibly still valid, in a technical sense, but may not make sense for the new provider component (*e.g.*, the new provider uses a different port or different configuration file). Using the old object with the new component may foul and crash it.

The question we are trying to resolve is: What happens to shared objects if the component that created the object disappears? The problem is shown graphically in Figure 33. The figure

shows that after a service invocation, there may be shared objects referenced by both components. When the component that created the object is invalidated, removed or destroyed (in this case this is the service provider component, it is unclear if the shared objects remain in a valid state or not.

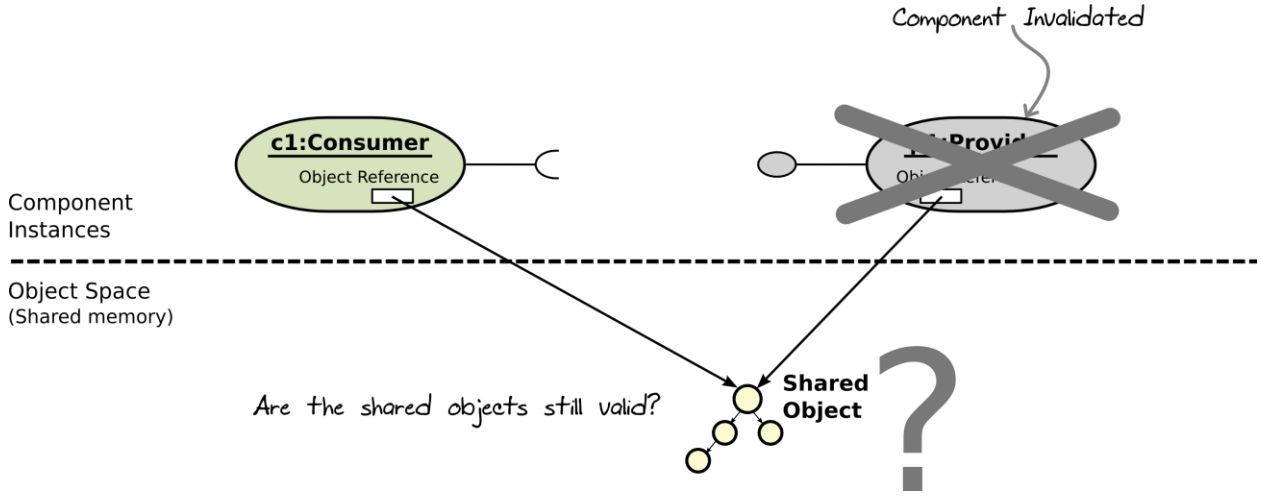

Figure 33: Shared objects referenced from two components require a mechanism to establish if the objects may continue to be used even after the component that has created them is removed.

Our approach is to augment the Service Contract concept with metadata to specify the retention policy for objects passed between components in a service interaction. We annotate types in the Service Interface (the initial point of contact between two components) in order to indicate if an object is Managed or Free. This adds a semantic dimension to the otherwise purely syntactic nature of the service interface. Annotations on parameters used in a method indicate the retention policy on objects created by the consumer and sent to the provider component. Annotations on return values are to indicate the retention policy used on objects sent from the provider component to the consumer[63,64]. This idea is very similar to *ownership types* [BOYAPATI et al. 2003].

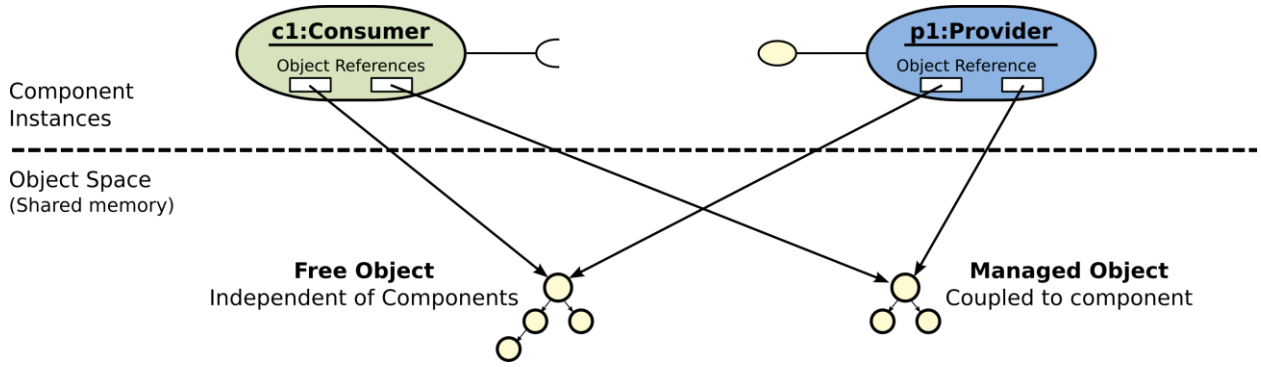

Figure 34: shows two shared objects with different retention policies. Free objects are independent of the creating component's lifecycle while managed objects become invalid if the creating component is invalid.

[63] Our model supposes that parameter objects are created by service consumer components and return value objects are created by service provider components. The term create is used sparingly; however, even if the component did not create the object it is responsible for passing it in either a parameter or return value to the other component.

[64] We do not support in-out or out parameters available in languages such as C++ or Ada. Even programming languages like Java can subvert the lack of in-out variables by using container or wrapper objects, with values set by the receiving components. These practices should be avoided because it breaks the directional nature of the service.

Figure 34 shows that components can simultaneously reference Free Objects, which do not have retention conditions, and Managed Objects, which components must pay particular attention to releasing when the component that created the object is invalidated.

Handling Managed objects requires releasing the object when the component that provided it in the service interaction becomes invalid. Figure 35 shows an example of a component that is removed. References from the consumer component to the managed object must be released, allowing the managed object to be garbage collected. If the reference is not released, this results in a dangling reference, potentially causing memory leaks, unexpected behavior, or failure. Note that the framework can force the release of managed objects by destroying the component (causing the garbage collector to eventually release the objects); however, it is not possible to easily detect such cases without extensive runtime instrumentation.

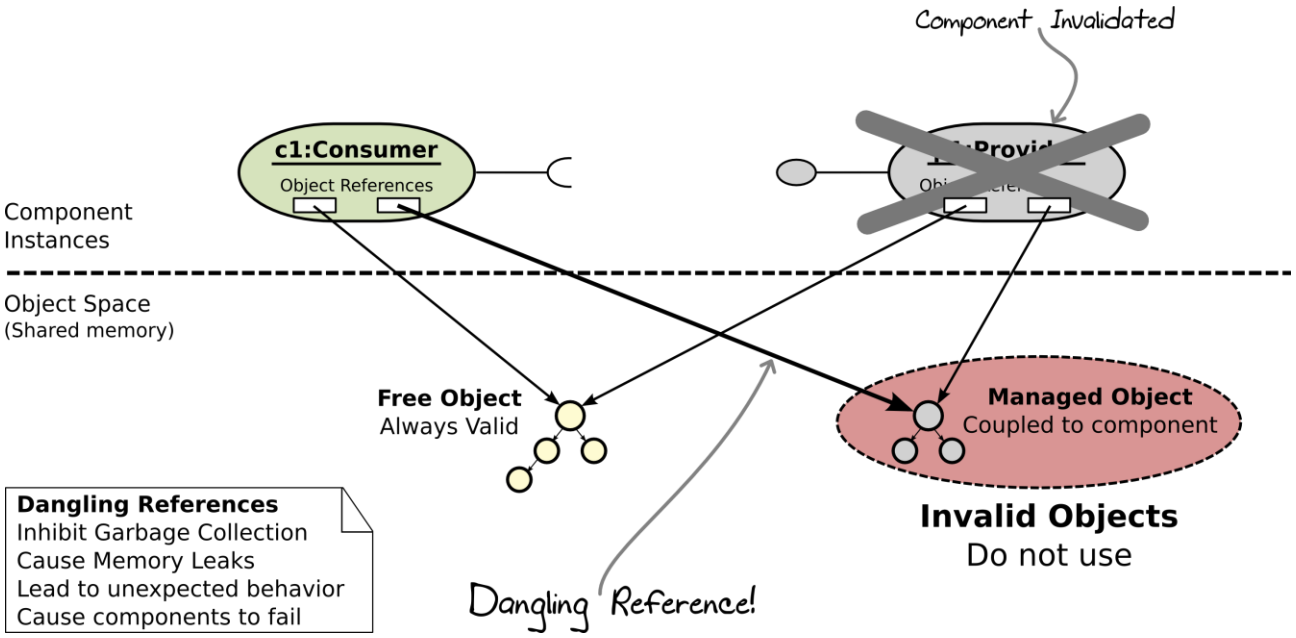

Figure 35: An invalid component can cause Managed Objects that to become invalid, leaving components that continue to reference these objects in a potentially corrupt state because of dangling references.

When programming a component that has multiple dependencies to the same service type, handling managed objects can become complicated—especially when the component is bound to multiple components simultaneously—because it would require tracking Managed objects along with the service dependency. Using dependency injection mechanisms complicates this issue even more given that dependency injection is often transparent to the component's code. Indeed, using method callbacks for binding components is easier because the service dependency reference can be stored in a container object (*e.g.*, a map) with a list of managed objects coming from that particular service. A map of service dependency objects that points to a map of managed per-service objects, or a new wrapper object created by the component to hold both the service dependency and the managed objects, are probably the easiest way to handle this case.

Finally, unlike Managed return value objects, Managed parameters are particularly difficult because the service provider, which must release the object when the service consumer component becomes invalid, is never directly bound to the consumer through a dependency (the consumer is

bound the provider) and does not generally distinguish between one consumer and another[65]. An identification mechanism is needed to distinguish between components and then to notify the component to release all managed objects held using the given identification. In order to avoid polluting the method call with parameters used for identification purposes, our approach uses thread local variables to pass the consumer component's internal identification[66] to the provider, which can then use it to construct its managed object reference map. When a consumer component is invalidated or removed, the provider component's callback method is invoked using the consumer components id. The id can be checked and managed objects released.

6.2.3 Characteristics of Free objects

Free objects should be by far the more common type of retention policy used in the Service Interface for shared objects. Free objects are conceptually similar to data that passes between nodes in a distributed system: the node can safely use the data as long as it needs to and the data is not spontaneously invalidated. Certainly, free objects are used just as Data Transfer Objects (DTOs) are for sending information between two components. However, they can be "complex" objects[67] too. As we have seen in our approach to implementation decoupling (see section 6.1 Decoupling component implementations), in centralized systems, components have a need to encapsulate and hide implementation details. Free objects allow for complex objects as long as the object is not coupled to the component that has created and provided it.

Free objects are recommended to be immutable and serializable. This is important because this can reassure the component that has a reference to the object that no other components will change the values or the state of the object in question. Indeed, immutable data is much easier to use in multi-threaded applications and adds to the scalability of the overall system. Serializability allows for the co-location or transparent distribution of the components[68].

Free objects abide by the following guidelines:

- Free objects are decoupled from all components and can transit freely throughout the application. They can be shared by various components.
- Free objects remain valid even if the creator component is invalidated.
- Most free objects should be immutable objects (*e.g.*, DTOs) (primitive data objects where there is no retention plan are always valid and useable)

[65] Most component models do not add support for providers to distinguish between consumers. However, in many cases a provider may use the current thread ID to achieve a similar result, since every thread has a unique ID. Nevertheless, the thread ID is not sufficient in our case for releasing Managed objects because different threads may be used by the same component, or the same thread used by different components.

[66] The identification value is already used by APAM in order to distinguish component instances. It is made visible to the components in certain cases, such as when releasing managed objects.

[67] We use the term complex object to contrast with Data Transfer Object, in the sense that, the object is not limited to containing data (generally primitive values only in distributed systems). Complex objects exploit the benefits of object-oriented systems by wrapping both data and operations into a single unit (namely, a class).

[68] Distribution concerns are not addressed by our work. However, given the benefit of serializability for such systems, there is added benefit to considering it when possible.

- Free objects can be complex objects if they do not interact with their creator component's internal state.
- Free objects use non-accounted resources (accounted resources are limited, and if limited they must be released, thus, they are no longer decoupled and must be managed).
- To further improve decoupling, Free objects should be contained in the Extended Service Contract modules in order for them to continue to be useable even when a component implementation is uninstalled. This is however an orthogonal concern.

6.2.4 Characteristics of Managed objects

Managed objects are used for shared objects that require special handling. Managed objects generally are used for objects that use accounted resources, such as thread objects, database connection objects, device access objects, among others. Because the underlying virtual machine cannot generally automatically or transparently handle objects that use accounted resources, there is a procedure to follow, these objects cannot be freely shared or passed from component to component in the architecture.

Another kind of Managed object are those objects used to store mutable state that is shared between consumer and provider components, like session state, a history of previous interactions, an agreement for a common protocol, an service-level agreement, or a common configuration.

It is difficult to find an equivalent to resource-based Managed objects in distributed systems because current commonly used approaches do not rely on such a concept, particularly given the level of coupling that this could introduce. In centralized systems on the other hand, the idea of sharing accounted resources between components is quite common. For example, a Java Enterprise Edition server may create close to a hundred thread objects[69] that are placed in various thread pools, waiting to be used to service operations. The same occurs with database connection objects, which are objectively more expensive. It's also interesting to note that in languages that use memory allocation mechanisms, where memory must be reclaimed manually, any object that claims memory has to be managed properly in order for it to be released[70].

State-based Managed objects can be found in distributed systems, under the form of special parameters passed back and forth between consumers and providers, such as session ID or transaction ID objects. As with centralized systems, if the provider is replaced by another one, passing the old Managed object in a parameter may result in unpredictable behavior, errors or a crash.

[69] This is the case of the JOnAS application server which has a thread pool for EasyBeans, one for Tomcat, one for Joram, among others.

[70] It is arguable to say that such programming environments are not ideal for creating centralized component-based dynamic applications. Indeed, sharing objects between components in environments that do not have a garbage collector would add to the programming burden and increase the risk of improper memory management and memory leaks. In such cases, the fallback is to rely on simple primitive data types instead of objects (which is the case for loosely coupled distributed systems), trading off flexibility and productivity for simplicity and decoupling.

Managed objects abide by the following guidelines:

- Managed objects are coupled to the component that created them.
  - They become invalid if the creator component is invalidated or destroyed.
- They are objects that may interact with the creator component's internal state.
- They are used for accounted resources.
  - Resource objects that need to be controlled or pooled (*E.g.*, JDBC connectors, threads).
- Special data objects that only make sense to use if the component that produced them is still valid.
  - *E.g.*, Configuration objects.

Components that receive a reference to a Managed object must be programmed to release the reference when the creating component is invalidated. To do so, it is expected that the component implement a notification callback method that is invoked by the framework when the creator component is invalidated. The creator component's identification number is used to distinguish between components when required.

6.2.5 Free and Managed objects and their implications on the service contract and encapsulation

Free objects increase the restrictions on how an object should be programmed by its providers, but it frees users of the object from having to deal with any special considerations regarding dynamism. Indeed, programmers should make free objects thread-safe, immutable[71] and serializable, allowing components to freely propagate the objects, use them or release them. These properties make the object inherently shareable.

The type of an object also has repercussions on encapsulation. Indeed, a component should no longer encapsulate the implementation classes of free objects because, should the component be removed, the free object would become invalid. However, thanks to the use of service extensions, free objects may be put into either the service contract or the service extensions, allowing it to be externalized from the component implementation.

Indeed, free objects must be capable of "outliving" the components that create them. Thus, their inclusion directly into the service contract is straight forward because service contracts outlive the components that depend on them. Including implementations of free objects into the service contract is a natural way of saying that "the effect of the interactions between two components is valid as long as the contract between them remains valid". Given that many components can rely on the same contract, and that the components may specialize the contract through service extensions, this is a relatively flexible solution to the compromise between decoupling and implementation hiding.

[71] Immutable objects are inherently thread-safe.

6.2.6 Notifications for releasing Managed objects

As described earlier, Managed objects need to handled and programmed in such a way as to ensure that, when the component that provided the object becomes unavailable, the object is no longer used by other components. We propose a programmatic approach to handling such cases. Indeed, the particularities of Managed objects make them difficult to handle in an automated fashion.

Moreover, Managed objects complicate the use of service dependency injection directly into a component's fields because, when dejecting a service reference, we must also notify the client to release references to the Managed objects. However, service injection and dejection is transparent to the component, while Managed objects are not. In theory, it is not mandatory to provide a notification service; the component that is not notified of an injection or dejection may continuously verify if the injected service reference is the same, and if not, it can release all Managed objects of the previous service. However, such a polling strategy is not practical.

The use of callback methods is much easier and straightforward for handling Managed objects. There are two different callback methods that can be used. The first one avoids the service injection issue by using a callback method for binding components. The component implements the callback methods `bind(Service serviceReference)` and `unbind(Service serviceReference)`, which receive a service reference indicating which service to bind and which one to unbind. In the case of multiple service dependencies, the service reference can be used to find the service in a list or container object held by the component itself. Similarly, using a second container object (probably a map), the component can save all Managed objects using the `serviceReference` as the key. This way, when the dependency becomes unavailable, the component can then find the list of Managed objects that were obtained from that component and release them.

The second method of callbacks is similar in essence, but is more general because it can be used by consumer components that use service injection and by provider components that receive managed objects from consumer components[72]. We provide a thread local variable that contains the previously executed component's internal identification number, which is already used by the framework for handling and identifying component instances. When a component obtains a reference to a Managed object, it can immediately check the thread and obtain the identification number of the component that created it. The framework ensures that the identification number is updated on every thread entry to and exit of a component. Components that require the previous component's identification number can recover the thread local variable using `String id = Component.getPreviousID().`

Finally, our approach to handling Managed objects is optional and does not need to be implemented. A component can be implemented with or without the necessary code to properly handle managed objects. When the component indicates that it releases managed objects by, for example, implementing a callback method, the framework will notify it when necessary and

[72] Provider components are referenced by consumer components and, as such, do not directly reference the components they are providing the service to. This makes it difficult for a provider to identify which objects should be released, if any.

suppose that the reference is properly released[73]. However, should the component not indicate that it handles managed objects, if the framework considers it possible for a managed object to be referenced, then, the framework may destroy the component in an attempt to force it to release the dangling reference[74] and to retain the application's consistency.

6.2.7 Coupling propagation: passing Managed objects

Handling managed objects requires not only releasing the objects when indicated by the framework that the creator component is no longer valid, but also paying attention to avoid propagating the object reference. Propagation is the receiving of an object and then passing it on to another component. Free objects do not have any particular conditions put on propagation, as by definition, they may be freely shared among components. However, managed objects must be properly released when the creator component is invalidated, otherwise, this may result in a memory leak or other undesirable behavior. Figure 36 shows that objects received through service interactions may be internally referenced and then sent to other components through successive service interactions.

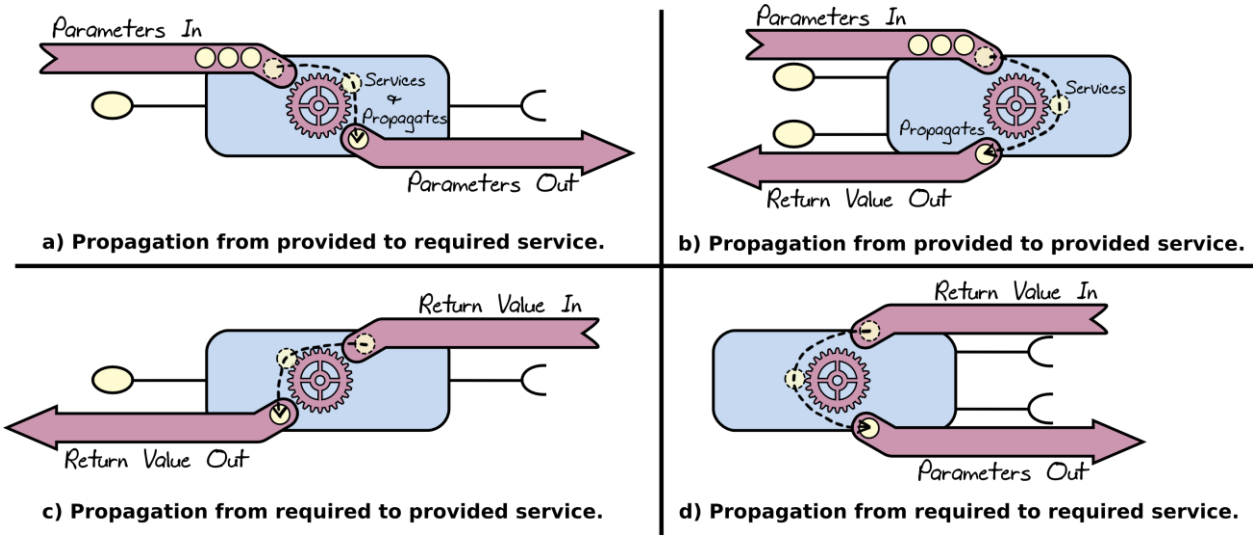

Figure 36: Propagation pathways in a component. A component may receive an object coming through provided or required services, and pass them to other components through provided or required services.

As can be seen, a component may pass objects in "any direction". In the case of a Managed object, a component may end up coupling other components by passing them Managed objects that they become unknowingly coupled to. Furthermore, because components are not obligated to be aware of Managed objects[75], they may (unknowingly to themselves) be passing Managed objects to other components. Indeed, one of the framework's tasks is to ensure consistency, and to do so,

[73] Indicating that the component behaves properly and releases managed objects when it does not is considered a software bug in the component.

[74] Fine-grain code-analysis may be used to indicate if a Managed object is in fact referenced and if that reference is held by the component or propagated to other components.

[75] Our approach relies on concentrating programming efforts for dynamism on critical parts of the architecture, while letting the framework ensure consistency in parts of the architecture that are not programmed for dynamism.

components that knowingly or unknowingly reference Managed objects must release them when necessary, by forceful destruction if needed.

There are two cases that the framework considers when analyzing potential propagation. The first one is when the component is aware of Managed objects and handles them properly. In this case, and only for Managed objects, propagation is indicated in the component's implementation class. Should the component receive a managed object and pass it to another component, the component indicates this information using either a metadata file or using annotations in its implementation class[76]. A simple example using annotations is given in Figure 37. As can be seen, the component implementation (which provides a simple proxy service) indicates propagation using, as origin, the Service type (*i.e.*, service interface) of the propagated managed objects[77], and as destination, the `Provides` service annotation. For components that are aware of Managed objects, the framework expects them to properly declare propagation. If this is not done, then the framework considers this to be a software bug in the component's implementation.

```
/* import headers */
import fr.imag.PrinterService;
@Provides (propagates=PrinterService)
public class PrinterProxy{
      @Requires PrinterService printer;//injected dependency

      /*method definitions start here*/
      PrinterConfigurator getConfigurator(){
            return printer.getConfigurator();
      }
//…
}
```

Figure 37: Component implementation example showing how to indicate Managed object propagation from a required service to a provided service.

The second case of propagation is with components that are not aware of Managed objects. If propagation is not handled, the worst case scenario is that the component propagates managed objects through all services it requires and provides, thus, transparently coupling all components it interacts with. In this case, once coupling becomes transparent and hidden to components, it is not possible for the components to properly release the managed objects[78]. Indeed, transparent

[76] The independence between two different Service Contracts means that we cannot specify this information in the Service Interface because some component implementations may propagate Managed objects while others do not.

[77] The limitation of using the Service Interface as that propagation is calculated using implementations and not instances, thus, it is less fine-grain. However, the advantage is that it is simpler to use, and it opens up the possibility of using static code analysis on propagation at design-time.

[78] Because components use the Service Interface to know which objects to properly handle (i.e., using the @Managed annotation) and they use Map objects to identify which components have provided the Managed objects, if a

propagation of managed objects may lead to the forced destruction of all components reachable from the propagating component.

Component destruction is not problematic for components that have been identified by the architect at design-time; however, transparent propagation is still problematic because components that are programmed to be aware of dynamism may not know they are receiving a propagated Managed object. Indeed, there is a lack of a protection barrier against such cases. Such a barrier is needed in order to ensure that a component will never hold a dangling reference caused by a hidden Managed object.

The only safe attitude, from the framework's perspective, is to calculate the contamination caused by the hidden propagation of Managed objects. This is discussed below.

6.2.7.1 Improving propagation analysis

There are multiple ways to improve analysis regarding the propagation of Managed objects. Using the Service Contract (*i.e.*, the Service's implementation classes), we can compare if a class that defines the Managed Object is reachable through other Service Contracts with which an unaware component interacts[79]. If it is not a reachable class (*i.e.*, the class is not contained in the transitive closure of reachable classes of the second service's Service Interface), then it is impossible for it to be propagated through that service. Such a calculation needs to be done for all services the component interacts with (both consumed and provided). However, if two service contracts have an overlapping class that is used to define a Managed object, then propagation is possible. If propagation is possible, the next calculation is to verify if propagation is through a Free object. If not, that means that the component transparently propagates a Managed object through an explicitly declared Managed object, and is thus no harm to other components. On the other hand, if it propagates the Managed object through a Free object, propagation becomes transparent and transitively couples other components. Because of the transitive nature of propagation, a contaminated component (*i.e.*, a component that has received a hidden managed object) must be analyzed using the same procedure to see if contamination is further propagated.

Static code analysis is another way of analyzing propagation. The idea is to analyze the component's source code to effectively deduce if propagation occurs. We explain this approach in more detail in section 8.3.4.

6.2.7.2 Design-time considerations of propagation analysis

In general, a component that is not programmed for handling dynamism should never be allowed to interact with services that define managed objects. This can be easily verified: if the component requires or provides a service that declares managed objects (*i.e.*, the service interface has a Managed object annotation), it should implement a notification callback method and handle

component receives a hidden Managed object that is transparently propagated by an unaware propagating component, the receiving component will not be expecting the notification to decouple from such an object even if the Framework should send one.

[79] Note that the transparent propagation of Managed objects only occurs through components that receive Managed objects (they are defined in the Service Interface) and are unaware of they must release them or indicate they propagate them (i.e., they do not receive notifications from the framework to release them nor do they specify propagation).

them correctly. If it does not, the designer should be warned and the component should be changed.

Should a component receive a Managed object and not be aware of notifications, it should be verified that the component does not propagate the managed object. If propagation does occur, the propagation path should be determined and the object through which propagation occurs in the destination Service Interface should be identified. The destination Service should have its Service Interface's object declared as a Managed object (using @Managed annotation) allowing components that use that service to be aware of coupling. Thus, even though the propagating component is unaware of propagation, other components can still be aware because the Service Contracts properly inform them of Managed objects.

Finally, particular attention needs to be paid to components that interact with Managed objects because of the risk of propagating the objects. When designing services, architects and developers should be attentive to identifying Managed objects in order for verification processes to be effective. When transparent propagation occurs, it can be extremely hard to identify how far the objects have been propagated, potentially causing large parts of the application to be invalidated in an effort to ensure consistency and avoid dangling references.

6.3 Conclusion

There are multiple facets to designing dynamic applications. In this chapter we have seen one critical aspect of dynamism: decoupling components in order to ensure correct dynamic behavior. Decoupling takes place at the component implementation and component instance levels.

Decoupling component implementations consists of modularizing components and services into individual units that can be installed, removed, updated and substituted at runtime without impacting other component implementations. Our approach is service-centric, where decoupling occurs around the service. We identify the Service Contract, which consists of the Service Interface and all transitively reachable types from the Service Interface. This approach allows for multiple provider and consumer components to use the same Service Contract and be bound together in any configuration, while still avoiding service incompatibilities, such as Class Cast Exceptions. Moreover, components can specialize the Service Contract by means of Service Contract Extensions. Specialization allows components to provide and require unique services while (partially) maintaining implementation hiding and encapsulation, an important aspect to building centralized applications.

Decoupling component instances describes how certain services may use objects that require special management. Dynamism means that these objects, generally never expected to become unavailable in a monolithic application, have to be handled differently. Our approach is to use an object retention policy that is declared in the Service Interface. We propose two policies: Free and Managed. Free objects can be used and shared however a component wishes, while Managed objects need to respect the retention policy. When the creator component of a Managed object becomes unavailable, references to the Managed object must be released. When designing a service, Free objects should be designed to be thread-safe, immutable and serializable to increase

the applications robustness. Managed objects should be attentively chosen and used for resources that require accounting and recovery (*e.g.*, any objects generally pooled like threads or database connections which are often not trivial to construct or destroy).

The most important argument of this chapter has been to show that developing dynamic-decoupled components—components used in the construction of dynamic applications—is a very difficult task filled with subtle pitfalls that can lead to unexpected and undesirable behavior if not cared for. The process of building complex applications built from dynamic and non-dynamic components can only safely be undertaken if programmers and architects are supported by tools and frameworks dedicated to this task. We argue that this support is required at all levels of software design and execution:

- Architectural level, to determine the dynamic zones of the application and to identify which components require dynamism and best benefit the application,
- Component level, to properly decouple component instances and create dynamic-resilient dependencies,
- Deployment and packaging level, to identify and decouple the modules required for the Service Contract, the Extended Service Contract and the component implementations,
- The framework, for managing the application's dynamism given expected and unexpected dynamic change events, all the while ensuring consistency given the level of decoupling among the components.

This support is required to safely develop robust dynamic applications.

Chapter 7

Dynamic Applications: Runtime Support and Consistency Analysis

"In our profession, precision and perfection are not a dispensable luxury, but a simple necessity."
—Niklaus Wirth, "A Few Words with Niklaus Wirth", Dr Carlo Pescio (June 1997)

Dynamic applications allow change. They are dynamic because components of the application can change and evolve at runtime without having to restart the entire application. Dynamic applications are the result of a shift from monolithic software architectures to Dynamic Software Architectures (DSA) composed of loosely-coupled collaborating units, which we call components. Previously in this dissertation, we have described how to construct such units and how design decisions taken at multiple abstraction levels (*i.e.*, at the module level, the component implementation level and the component instance level) affect and are affected by changes in the application's architecture. Notably, we have focused on decoupling components so that change impact will be minimal and isolated to individual components.

In this chapter we will focus on the application's architecture at runtime and how to ensure that its components remain consistent. We consider an architecture to be consistent if all its components are consistent. However, small architectural changes at runtime (at both the implementation and instance levels) can cause domino effects across the architecture and lead to large parts of the application being invalidated, which is why decoupling is so important. Nevertheless, decoupling components can be difficult and potentially costly, which is why our approach proposes decoupling techniques to ensure consistency but does not force them upon developers or architects. On the contrary, our approach is—given any level of coupling among the components—to minimize the impact of changes on the architecture at runtime while continuing to ensure the application's consistency. Which components to decouple to better handle dynamism becomes a design decision.

Furthermore, unexpected dynamism (*e.g.*, bugs, failures or forced changes) and component volatility are critical elements in the design of robust applications. Failure, which is volatile and unexpected, is particularly difficult to handle and requires isolation barriers (that can affect a component's business logic) to ensure consistency. Calculating components that are potentially corrupt and removing them from the architecture ensures consistency. Ensuring consistency is crucial to enabling dynamic applications.

7.1 Failure detection

Detecting when and where a failure has occurred can be problematic, but our approach requires finding the point-of-failure in order to calculate the degree of collateral corruption and to perform the necessary reconfigurations. This section briefly describes what we mean by failure and how the detection is performed.

We have described why it is unrealistic to have to identify all components that can fail in order to make the application failure-resistant (see section 4.7.1). Without a proper mechanism to decide what constitutes a component failure and when the failure invalidates the component, there is effectively little failure-tolerance except for failure originating in volatile components. Indeed, precisely identifying when and where a failure occurs and, consequently, if the failure requires an architectural change to recover, is highly desirable. This work has not focused on failure identification, which itself is a potentially extensive subject of its own, yet there is much overlap between failures and our tolerance to volatility.

Section 5.1 describes components' expected dynamic behavior. Dynamic behavior can be stable, detachable and volatile. The underlying notion behind volatile is that, once it becomes unavailable, it can no longer be used and any active requests need to be aborted. We expect that developers of volatile components use an exception, namely, the `ServiceUnavailableException`, to abort any active requests. Moreover, this exception tells the framework that the component has become unavailable (*i.e.*, has failed), and that the component is no longer useable.

The generic exception mechanism (*i.e.*, the `ServiceUnavailableException`) to indicate that "something" has happened and that an "architectural reconfiguration is required" can be used by components of any dynamic behavior to indicate that they have failed. The detection of these exceptions occurs at the component's frontiers (*i.e.*, provided and required services) and is otherwise not intrusive. However, this continues to fall back on the notion of "failure-able" and "failure aware" components because the components know and throw the exception. This is natural for volatile components that know they might fail, but not so much for other components.

In Table 2 we provide a comparison of the different dynamic behaviors and how they relate to the activeness of the dynamic change (*i.e.*, reactive or proactive) and the expectedness of the dynamic change. As we can see, the unexpected reactive dynamism includes component failures.

		Expectation of dynamism	
		Expected dynamism	***Unexpected dynamism***
Activeness of change	***Proactive change***	Detachable components	Forced update (only applies to stable components)
	Reactive change	Volatile components	Failures, software bugs, …

Table 2: Activeness of change compared to expectation of dynamism

A natural first intuition would be to extend the exception mechanism to simply include all unchecked exceptions (*e.g.*, all runtime exceptions) and treat them identically to the `ServiceUnavailableException`. This approach has its drawbacks because not all unchecked exceptions are used to indicate failure. Indeed, we cannot always suppose that an exception that crosses component boundaries is indeed a failure. The line between deciding to use unchecked and checked exceptions can, in fact, be very thin. In order to avoid overreaching with our failure detection mechanism, we provide a configurable mechanism, where the architect decides which exceptions are treated at the architectural level (*i.e.*, wrapped into a `ServiceUnavailableException`) and which exceptions are left to the application's logic to be handled directly by the components. Interestingly, even some checked exceptions might simply be treated as an architectural error, allowing components to not have to implement the try-catch-finally clause or the potentially complicated logic that they often require to recover.

The result of such an approach on the application's dynamism is worth mentioning. Depending on how the architect decides to handle exceptions, we have:

- Empty exception list: Only the ServiceUnavailableException is used to cause architectural reconfigurations. Only explicit failures are handled by our mechanism and cause reconfiguration; any other exception is expected to be handled by the application's logic without reconfiguration.
- Selective exception list: Provides a list of exceptions that are considered failures that are equivalent to the ServiceUnavailableException.
- All inclusive list: Any and all unchecked exceptions that are caught at the component's frontier are considered equivalent to failure and treated as a ServiceUnavailableException.

We believe this mechanism is flexible enough to treat a large array of common failures in current systems, and because it is configurable it avoids the risk of overreaching. Of course, detecting exceptions is still a limited way of detecting failure. A more complete failure detection mechanism is complementary to our approach and can easily be included. Such a mechanism can use much more complete analysis, for example, component input/output analysis, to decide when the component must be replaced by throwing the ServiceUnavailableException. The rest of our approach can remain intact.

The following sections focus on calculating the extent of potential corruption caused by dynamism and on recovering from dynamic changes, including unexpected dynamism and component failure. They implicitly rely on the detection of the point-of-failure or, more generally, the point-of-change we have described.

7.2 Minimizing recovery using isolation barriers

Programming dynamic applications raises the question of how far we can transparently handle dynamism for developers and architects. The goal of many projects is to allow us to program components that are oblivious to dynamism, and yet paradoxically, still support change. Some approaches have shown that there is a tradeoff between programming model and the support for transparent dynamism[80]: the more restrictions placed on the programming model the more guarantees can be supported[81].

When attempting to minimize the restrictions on the programming model, and specifically for centralized component-based applications that allow sharing complex objects, there are many aspects of dynamism that can be automated, externalized and managed transparently; however our experience and results support the conclusion that dynamism cannot be fully transparent. Indeed, we have shown that decoupling components is not entirely transparent to neither programming (*i.e.*, the source code reflects instance decoupling concerns) nor modularization (*i.e.*, classes are packaged in order to construct independent modules). Failure is another concern that is not transparent when programming dynamic components.

Component unavailability in centralized applications is different from distributed applications because there is generally not an interruptible communication medium between the components (*i.e.*, the network). This has led to approaches that continue to use a component even when the component is being removed[82]; we call this *service caching*. Such approaches generally omit the fact that components can and do fail, or that components can act as proxies to distributed services or to physical devices that can be interrupted. Indeed, just because we hold a reference to a component does not mean that the component is useable.

We have introduced this cache vs. fail dilemma by characterizing components according to their expected dynamic behavior: namely, *detachable* components and *volatile* components. Detachable components allow service caching (they remain useable and allow finishing current operations[83]), while volatile components are used to express immediate unavailability (similar to failure, they cannot service any further operations and current operations on them must be aborted). Volatile components are used to represent devices and remote services. Component volatility is a step towards recovering from component failures (potentially any component can fail

[80] For example, cloud computing approaches restrict, among other things, how state can be handled. These restrictions decrease coupling and increase the application's potential for scalability.

[81] In some approaches the programming model is not explicit in the source code (*e.g.*, POJO based approaches) but is still present in the approach's underlying philosophy. A general set of rules or programming restrictions often needs to be followed when programming these components in dynamic environments.

[82] Current operations being service by a component are allowed to finish while new operations are initiated on other components.

[83] Robusta exploits this feature at runtime by allowing current operations to finish without failing.

at runtime)[84]. In addition, component volatility can propagate through the architecture causing potentially any component to fail if not properly protected. Indeed, tolerating components that can fail introduces a large array of issues that need to be addressed, but it also increases the number and types of applications that can be constructed using the platform.

To mitigate the propagation effects of a failure, we introduce the concepts of isolation barrier and recovery mechanisms. Isolation barriers contribute to system resilience by providing **failure boundaries** permitting part of a system to fail without compromising the whole [Aiken *et al.* 2006]. Isolation barriers are implemented by consumer components to guard against possible corruption caused by the service provider's failure. Recovery is the process through which a component recovers from a failure and continues proper execution from a consistent state. We characterize isolation barriers and recovery mechanisms according to the consistency guarantees they allow and the level of intrusion they have on the component that implements them.

7.2.1 Localized recovery

Localized recovery is the process of repairing the architecture as close to the point of failure as possible. Figure 38 shows the implementation class of a simple consumer component in a simple architecture. The architecture is also shown, composed of two components, a consumer and a provider component. The sample code shows that the dependency is injected into the consumer component using the `MyService` field.

```
@Component
public class MyClass {
    @Requires
    MyService service;

//[...]
    public void usefulMethod() {
        service.call(obj1, obj2);
    }
}
```

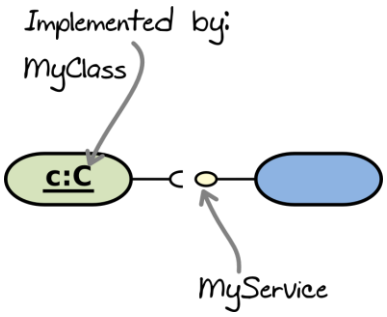

Figure 38: Describes a simple architecture and the consumer component's implementation class.

Supposing that the provider component fails, it is essential to be sure that the consumer component remains valid and consistent before finding another suitable service provider for the desired service. Figure 39 describes this process graphically. When the provider component becomes invalid, we need to know if the consumer component is consistent. For the consumer component to be consistent, it must be decoupled from the provider (*i.e.*, instance decoupling) and the failure caused by the provider component must not have corrupted it.

[84] Component failure remains more difficult than component volatility. Indeed, our work does not address failure characterizations or how to properly identify a failed component.

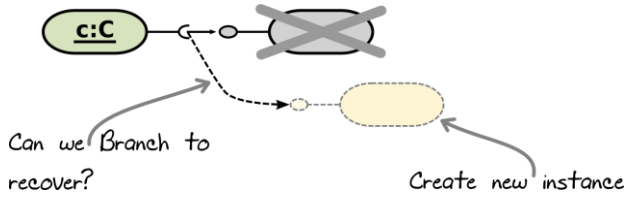

Figure 39: Branching in a simple architecture in case the provider component becomes invalid.

Informally, we call this branching. Branching is the process of finding an alternative solution to a failed or invalid component and rebinding the architecture to the new component as close as possible to the failure.

Localized recovery is part of a larger analysis on the architecture. Once a component fails, we need to calculate if any surrounding components are corrupted because of that failure, which in turn can further propagate corruption. A single failure can cause a large part of the architecture to become invalid in a domino effect, as described in Figure 40. Corruption can be propagated, causing more corruption.

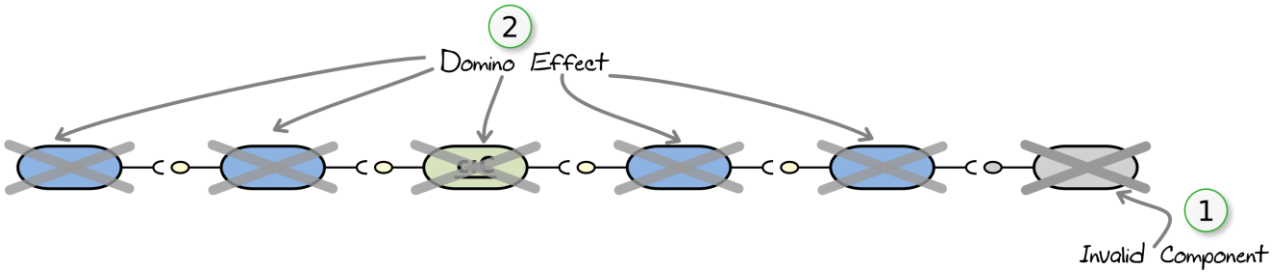

Figure 40: A single component failure can cause many components to become invalid.

Finding a branching solution in a larger architecture has to consider which components have been corrupted, remove them, and rebind valid components as close to the point of failure as possible. The problem of finding where to branch is shown in Figure 41.

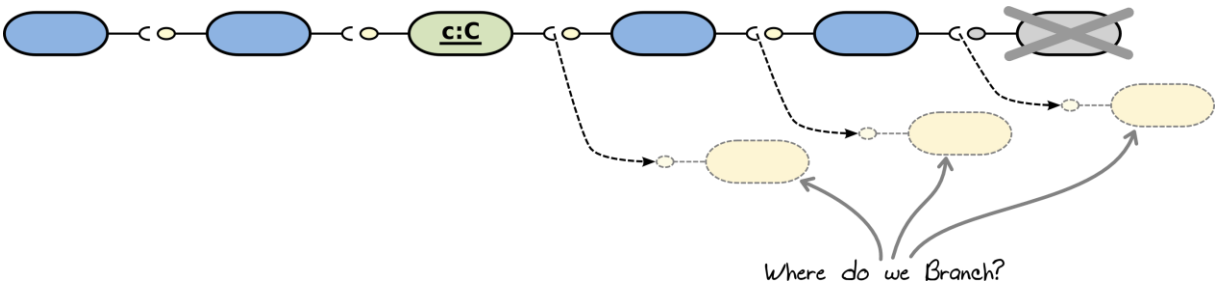

Figure 41: A single failure can cause various components to become invalid. The framework must find a branching solution that ensures consistency and minimizes the impact of the failure.

In order to guarantee that a component has not been corrupted, and thus, allow branching, we use isolation barriers and a recovery mechanism to ensure consistency. These mechanisms allow selecting key components, which become flex points in the architecture and allow rebinding to new components should a failure occur. These mechanisms are described next.

7.2.2 Isolation barriers and recovery mechanisms

We use an exception-based mechanism to express that a service invocation has failed allowing the component to recover and return to a consistent state. Exceptions are straightforward because developers are already accustomed to using them. A service failure, *e.g.*, caused by an unavailable volatile component, causes an unchecked runtime exception to be thrown by the framework upon the component invoking the invalid service. Runtime exceptions do not have to be declared and are thrown when an unexpected condition has occurred[85]. The use of unchecked exceptions means that the compiler does not force a component to write exception handlers to catch the exceptions, allowing components to remain unaware of failure if desired. Furthermore, components that wish to be aware of dynamism can simply implement a `try-catch-finally` clause to manage the exception.

The process of isolation and recovery consists of catching and managing the exceptions caused by failure, and returning the component to a consistent state. We call this the recovery process. Both isolation and recovery can be intrusive (directly implemented in the component's business logic). Depending on the level of isolation and intrusiveness, we propose the following recovery mechanisms: *none*, *external* and *application-specific* recovery. The recovery mechanisms are used to ensure that corruption does not occur.

7.2.2.1 No recovery mechanism

The inexistence of an isolation and recovery mechanism means that the component does not catch or manage the exceptions caused by service failures. The nature of runtime exceptions means that we cannot be sure that the component remains valid unless it is resilient. Indeed, an unexpected exception can cause the component's invariants to become inconsistent, leaving the component in an inconsistent or invalid state (the component might be corrupted). This is shown in Figure 42.

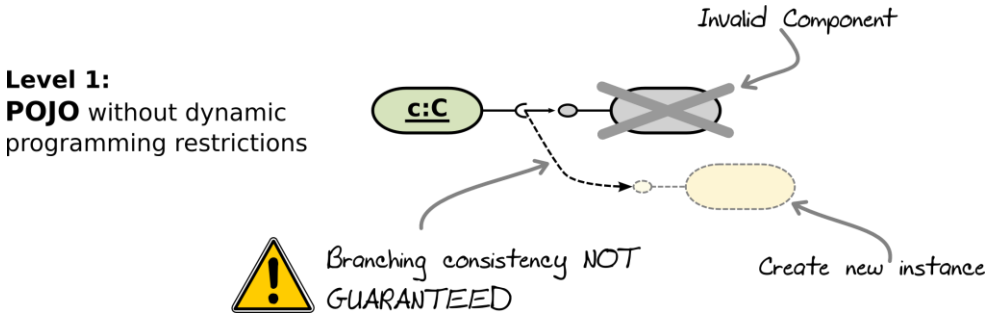

Figure 42: We cannot guarantee the consistency of a component that is not aware of dynamism, as is common in the case of POJO (Plain Old Java Object) approaches.

Components that do not implement resilient dependencies or a recovery mechanism are considered to be programmed unaware of dynamism. They are particularly oblivious to the fact that a service provider may fail unexpectedly. In essence, this means that they cannot be trusted after a dynamic event. In order to ensure consistency, the framework must destroy the component

[85] Many such exceptions exist, as for example, accessing an out of bounds index on an array or invoking a method on an object in an incorrect state.

instance. This causes state-loss but is both simple to program and ensures consistency because new components start from a de-facto state of consistency.

There are several exceptions to the rule that, if indicated, are sufficient to ensure the component remains consistent even if there is no recovery mechanism:

- *Stateless component*: if the component does not have an internal state there is no reason to destroy it because there is no state to corrupt.
- *Immutable state*: if the component's state is not mutable, it is also not corruptible.
- *Incorruptible state*: the developer can indicate that the component is tolerant to exceptions (and thus corruption) caused by the failure of its dependencies. The framework will trust the developer's assumptions.

The lack of a recovery mechanism (or the lack of assurance that the component is resistant to dependency failures, see section 8.3.3) means that the framework must destroy the component because it is potentially inconsistent.

7.2.2.2 External recovery mechanism

Externalizing the recovery mechanism is possible under specific and well identified conditions. Particularly, the risk of handling the recovery mechanism externally is that the failed service invocation has corrupted the parameters passed from consumer to provider components. External recovery allows the framework to transparently re-bind the component at runtime and to invoke the new component using the previous parameters. Being external, this is transparent to the consumer component's business logic. As shown in Figure 43, this approach requires additional metadata in order to ensure that the parameters are consistent.

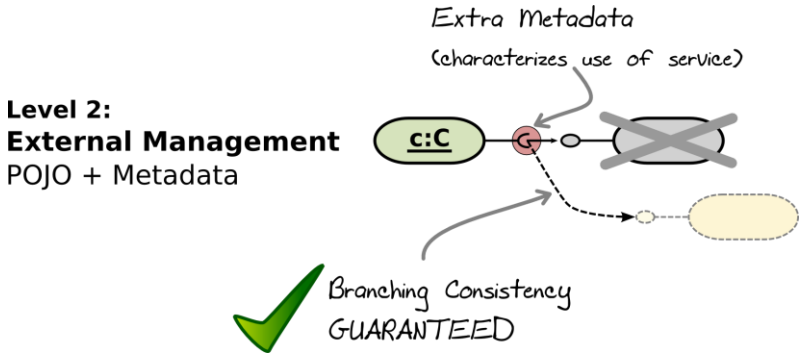

Figure 43: An externalized recovery management is possible if additional metadata is available to ensure that the service invocation has not been corrupted (the parameters used in the invocation are consistent and reusable for a second invocation).

Generally, in order to guarantee that the parameters remain consistent, they should be immutable objects. (In section 6.2 we described Free objects and recommend that they be immutable, serializable and thread-safe). However, detailed analysis of the provider component may show that it does not alter, or at a minimum, corrupt the parameters, such that the request can be serviced by the new component.

If it is determined that the request can be serviced by the new component, the consumer component is not notified of the change[86]. If it is determined that the request is no longer valid, the `ServiceUnavailableException` is thrown. The consumer component, if it does not internally catch the exception may be corrupted and will consequently be destroyed in order to ensure the application's consistency. Exceptions to requiring the destruction of the component are, as previously described, if the component is stateless, immutable or incorruptible.

7.2.2.3 Application-specific recovery mechanism

The consumer component can implement a customized recovery mechanism inside of its business logic. *Application-specific* recovery is achieved by wrapping every service invocation in a `try-catch-finally` clause that catches the `ServiceUnavailableException` thrown when a service failure occurs. This is presented in Figure 44.

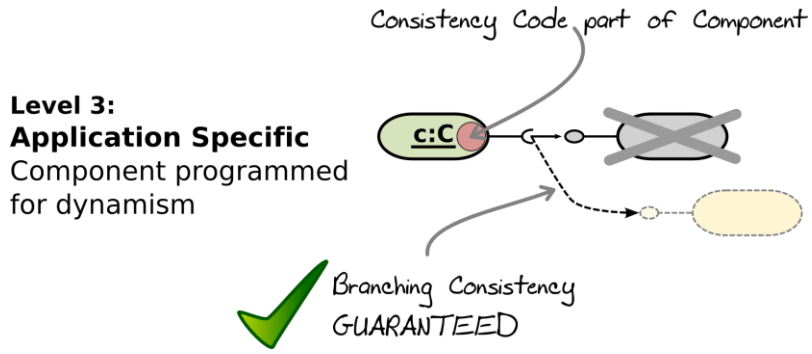

Figure 44: Application-specific recovery mechanism handles service failures inside application code to ensure consistency.

Fine-grained customized handling allows a component to use custom code to ensure consistency in more complicated scenarios. If consistency cannot be achieved, the component may proceed to shutting down and then throwing a `ServiceUnavailableException` to indicate that it has not achieved a consistent state or that it cannot complete the current request.

7.3 Application Consistency and Corruption Analysis

The objective of corruption analysis is to ensure that the application is consistent after a dynamic change. To do so, the framework must ensure that all of the components are consistent. Consistency implies that we must be sure that the components' states have not corrupted by

[86] It should be noted that, particularly in the case of externalized recovery, partially executing a request that fails, and then, re-executing the request using another provider component can lead to side-effects (*e.g.*, writing to a file, accessing a device, printing onscreen) that can become undetectable by the consumer component. If the component needs to be informed of a failed service invocation, it should use an application-specific recovery mechanism.

dynamism, that the state of open executions[87] have not been corrupted, and that the components' business logic has been properly respected. During this process, the consequences of a dynamic change event are analyzed in order to calculate, given current execution conditions, the components and executions that will potentially be corrupted by the change or the components and executions that have already been corrupted.

The consequences of a change depends on various factors, as, for example, the type of change at hand, the activeness of the change, component coupling, and open executions (*i.e.*, threads). Once the consequences, which we call the *change impact*, have been identified, the framework can search for branching points in the architecture to redirect new executions to replacement components. Branching must consider component decoupling, dependency resilience and open executions in order to be successful. Furthermore, because executions are continuously advancing through the components and their trajectory[88] is not known, this creates a level of unpredictability. This unpredictability leads to two branching techniques: *optimistic branching*, which supposes that the open execution will succeed and finish properly, and *pessimistic branching*, which supposes that the open execution is at risk and will fail. Interestingly, optimistic branching can fallback to pessimistic branching at the additional cost of removing the optimistic branches that failed. Both branching techniques are described in section 7.3.3.1.

Finally, our approach uses various optimizations in order to reduce the impact on open executions. Where possible, we allow branches to coexist, redirecting new executions to new components and reusing old components for old executions. This allows the framework to handle multiple architectural branches simultaneously (potentially transparently), while gracefully stopping and removing old components.

7.3.1 Activeness of dynamic change

It is important to distinguish the origin of a change event and the effect that such a change has on the consistency of the application. Changes can be proactive or reactive, and symmetrically, the process of ensuring consistency can be proactive or reactive. We explain both types of change and the consistency processes that follow them.

7.3.1.1 Reactive change and consistency

The application is reactive to change when dynamic change occurs bottom-up, directly from the components that form the architecture. Reactivity implies that the application has to change to take into account the event that has occurred. Such events are seen in, for example, components that fail at runtime (the application does not know they will fail, they just do). Our approach characterizes the dynamic behavior of components, such that reactive components declare their volatility (see section 5.1). Volatile components are, by their nature, causes of reactive changes in the architecture.

[87] Executions are mapped onto threads so this means that the thread's state must not be corrupted. The difference between component state and thread state is that the thread's state is transient. Furthermore, component state is stored in the component's fields, while thread state is stored on the stack.

[88] Trajectory is informally used to indicate that the components that will effectively be executed by the thread are not known a-priori.

In regards to consistency, a reactive change can cause the application to become inconsistent. This occurs because a component that fails or becomes unavailable (such as a physical device) can no longer be used, forcing the application to react. This means that a recovery process to return to a consistent state is required. Our approach to this is to detect and destroy components that are found to be potentially corrupt. Furthermore, we abort the executions that are also found to be potentially corrupt. Both of these processes lead to the propagation of corruption across the architecture, which can be mitigated using isolation barriers and recovery mechanisms that increase the resilience of a component's dependencies sufficiently to protect against such corruption (see section 7.2).

In general, any component that fails or is explicitly and forcefully changed results in a reactive change. Reactive changes are discovered at runtime when a `ServiceUnavailableException` is thrown. The framework uses the same exception mechanism as described earlier (see section 7.2) to detect components that are no longer useable[89]. It should be noted that, at design time, our proposal recommends declaring components that can fail unexpectedly as volatile components[90]. In general, stable and detachable components do not cause reactive changes.

7.3.1.2 Proactive change and consistency

Proactive change can be seen as occurring in a top-down fashion. A change order is sent to the framework, which then reconfigures the application to reflect the change. The important difference with regards to reactive change is that proactive changes can be controlled by the framework and should never lead to an inconsistent state. Indeed, because the framework controls and manages the change process, no executions should be aborted and only components that are not properly decoupled will be impacted by the change.

Proactive change avoids the situations that introduce inconsistency. At runtime, the framework can branch the architecture at a safe and desirable point, and redirect open executions to new components, while gracefully stopping and removing old components that are impacted by the change event.

7.3.2 The impact of change

To better understand how a change event can affect surrounding components, we describe how different types of change potentially corrupt components at runtime and how the corruption is propagated through the architecture. Furthermore, propagation depends not only on the components (*i.e.*, how they are programmed), but also on the current status of open executions in the architecture. Open executions that are impacted by a change and need to be aborted can further propagate corruption.

[89] Our efforts to detect component failures or malfunctions are limited to components that indicate there is a fault. We expect components that fail to throw the `ServiceUnavailableException`. This can be extended to use a more sophisticated and thorough process of failure detection. This is described in section 7.1, but remains outside of the scope of this dissertation.

[90] Although this information is not particularly useful at runtime because the framework uses exception detection to find reactive changes, it is very useful at design to ensure that there exist resilient dependencies sufficient to protect the application from reactive change events (or failures).

Seen from a high-level, the following types of change to the application's architecture are possible:

- Add a component implementation,
- Remove a component implementation (causes the removal of all its instances),
- Create a component instance (the component implementation must be installed),
- Remove a component instance.

Adding components, either implementations or instances, is not troublesome to consistency because all components are initially in a de-facto consistent state. Additionally, existing components are not impacted by adding instances or implementations because component-oriented approaches decouple components sufficiently to ensure what we call static-decoupling, which allows late binding (among other characteristics). However, removing components requires consistency checks to ensure that the remaining components are consistent. Removing a component instance may affect other component instances if they are coupled. Removing a component implementation, which is a process performed at the module level, may impact other component implementations because the modules are not properly decoupled. (Dynamic decoupling is discussed in Chapter 6.) Additionally, if a component implementation is removed or becomes invalid, all of its component instances also become invalid and should be destroyed.

Finally, open executions (*i.e.*, execution threads) in the application must be considered. When a component instance is invalidated, this may further invalidate partially executed operations (active threads) in the application. The framework may abort open executions that are potentially corrupt, causing corruption to spread further. The next sections analyze in more detail the effects of removing components.

7.3.2.1 Removing component implementations

Component implementations are packaged into modules. In section 6.1 we have described how to decouple component implementations using a five-module approach in order to decouple the Service Contract and Service Contract Extensions from the component implementation modules, allowing component implementations to be removed without impacting other implementations. This is a flexible tradeoff between the needs of decoupling and implementation hiding. Nevertheless, our approach to decoupling is a recommendation; we allow developers and architects to package implementations however they want. This allows efforts to be focused on areas of the application that require dynamism, instead of forcing all components to be decoupled. (The tradeoff is a loss of fine-grain dynamism caused by coupled implementations and instances.)

Modules are added and removed, and they represent the deployment units that reify dynamism at the implementation level. Modules contain, among other things, classes and interface that are used to construct component implementations. Furthermore, the Service Contract is also contained in modules.

Regarding the impact of removing a module, modules have two types of relationships: the *Depends* and *Extends* relationships. An *extends* relationship implies a depends relationship, and the impact at the module level is the same for both types of relationships. As shown in Figure 45, if a

module *depends* on another, and the dependee module is removed, the dependent module is invalidated. On the other hand, if the dependent module is removed, the dependee module remains valid (dependencies between modules are obviously directional). This assessment is straightforward.

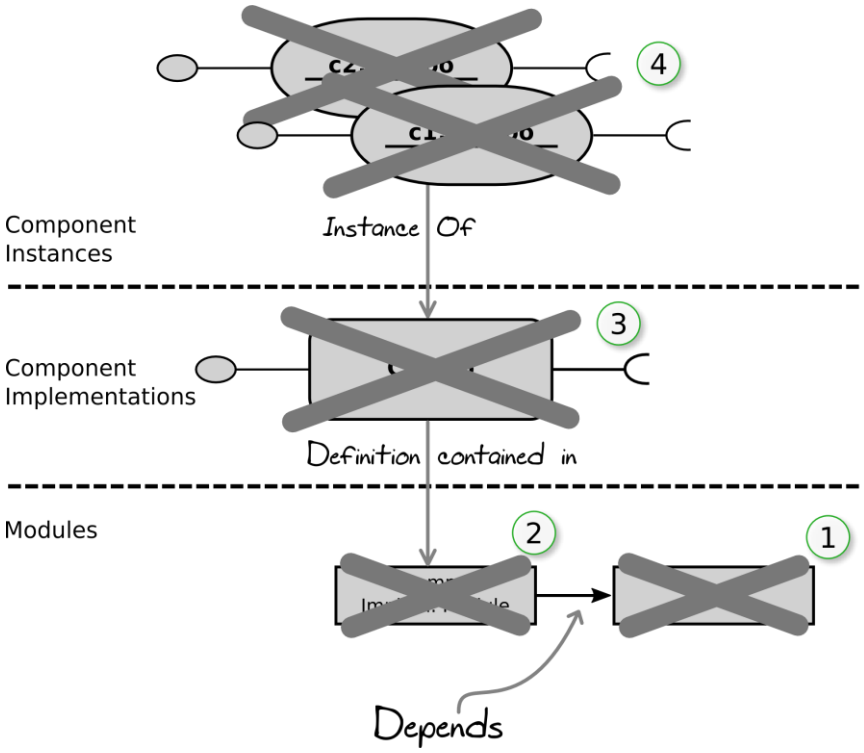

Figure 45: (1) Removing a module causes (2) dependent modules to be invalidated. (3) Component implementations contained in invalidated modules are also invalidated. (4) Component instances of invalidated implementations must be stopped and destroyed.

However, less intuitive is the fact that an *extends* relationship can cause component instances to become invalid even though there is no direct relationship between the modules. This occurs because extending a module, by either inheriting classes or implementing interfaces, means that the component instance that directly references classes in the extended module may actually be indirectly referencing classes contained in the extension module. This problem has been introduced from sections 6.1.2 through 6.1.4. Figure 46 shows the impact of removing an extension module on component instances and component implementations. It is particularly interesting to note that the independent component implementation is not invalidated when removing the extension because it does not directly reference any classes contained in that module. Nevertheless, the effect on the component instances is clear; they become invalid because they potentially indirectly reference classes in the modules being removed.

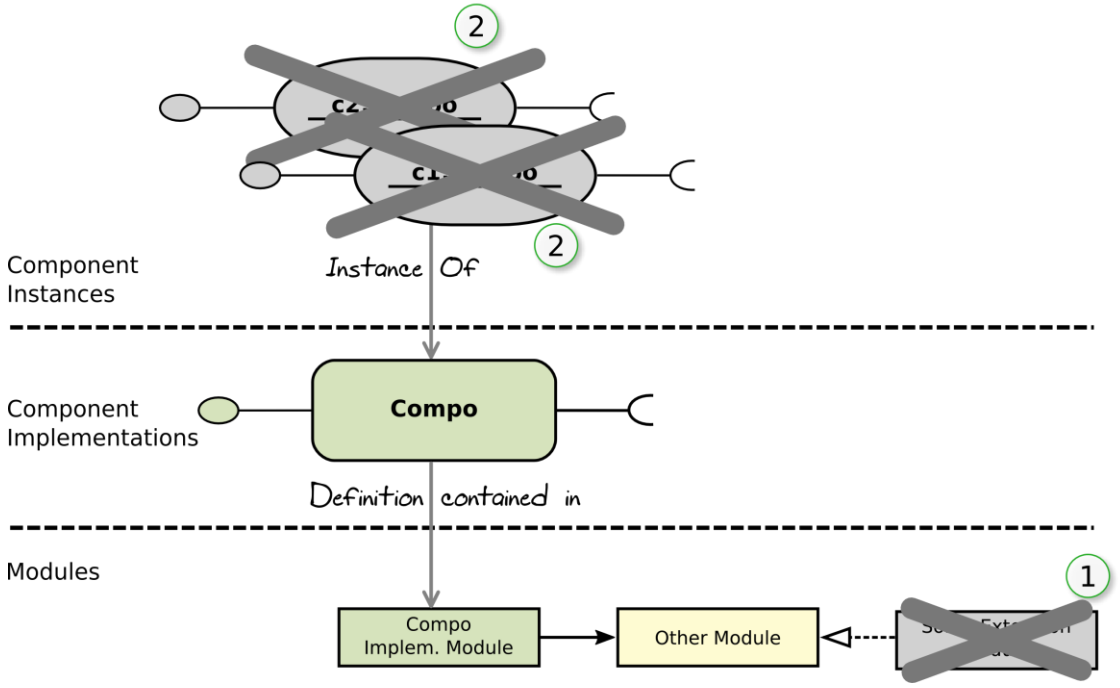

Figure 46: (1) Removing an extension module has little effect on independent modules, but (2) it invalidates component instances because they might indirectly reference objects defined by classes contained in the invalid module.

This may discourage the use of module extensions given the fact they can impact component instances in such a way. However, extension modules are necessary to allow implementation hiding (the extension module's classes are not known directly by the components[91]) and should be designed to remain installed in the system even though the component's implementation modules are removed. Furthermore, we have evoked the possibility of selectively invalidating component instances that have very possibly obtained references to objects of hidden classes in section 6.1.4.1, while not invalidating components that we are absolutely sure have not. Additionally, we have described how to lazily remove extension modules in section 6.1.4.2. These optimizations, and others we have yet to explore, mitigate the defects of using extensions, allowing developers and architects to exploit their benefits.

The next section details the effects of removing component instances. It is important to note that removing component instances does not impact modules or component implementations.

7.3.2.2 Removing component instances

The impact of removing a component instance varies depending on the relationship other instances have with it. At a minimal, removing an instance will affect the instances that are directly bound to it, causing them to be rebound to other components if possible[92]. Moreover, an instance can be inactive, meaning that there are no open executions that are using the instance, or active,

[91] This is important because this allows multiple components to provide implementation classes for the same interface, transparently. This also allows sub-typing classes to specialize behavior.

[92] Our work does not focus on selecting other components to bind with. This is left to the APAM framework which can choose which components to bind, can instantiate new components and can deploy component implementations, all in an attempt to satisfy a dependency.

meaning that it is currently servicing a request. Active instances are more complicated to treat because we must manage the threads and the thread state in addition to the internal state of the instance. Furthermore, removing a failed instance is different because any active executions must be aborted because they cannot properly finish (this occurs with volatile components that become abruptly unavailable). We continue this section by analyzing the different impact cases according to the relationship and activeness of an instance.

Decoupled component instances (instance decoupling is discussed in section 6.2 Decoupling component instances) can be individually removed and the remaining component can be rebound to another. This is shown in Figure 47.

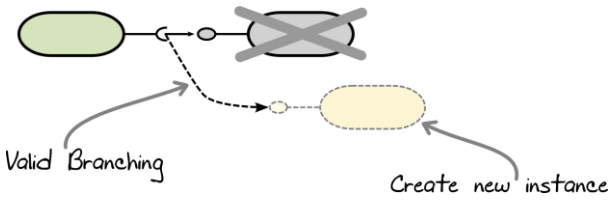

Figure 47: A decoupled component remains valid and can be rebound to another component if its provider is inactive and abruptly fails at runtime. Rebinding depends on finding a component that provides a compatible service.

Coupled component instances cannot be individually removed. Removing a component instance invalidates its coupled instances. This is because the coupled instance uses Managed objects that are not properly released[93]. The effects of removing a coupled instance are shown in Figure 48.

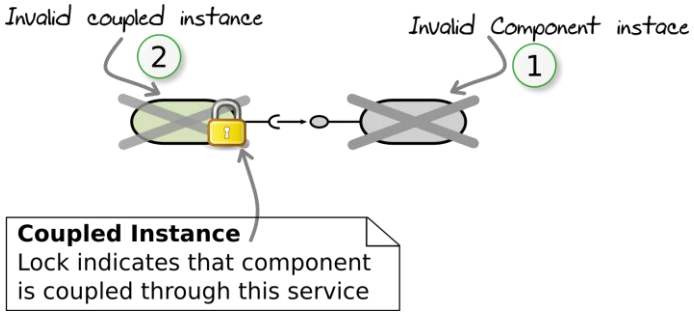

Figure 48: Removing a component instance that another component is coupled to causes the coupled component to become invalid.

Active component instances add the possibility of corrupting the current execution thread. If a removal is proactive, we can gracefully remove a component instance by passivating the instance (in order for it to stop initiating new requests) and allowing all open requests to finish before stopping the instance and destroying it. As long as the instances are decoupled the remaining instances are valid. If the instances are coupled, we passivate all of them because they cannot remain in a consistent state so there is no need to manage them individually.

[93] The term Managed object is used sparingly. It is possible that the components cannot be separated because their internal states become intimately coupled, or the component does not properly release accounted resources or other special objects. To ensure that the components' states do not become coupled, we propose using Free objects for service interactions because they are guaranteed to be independent of the component that has created them. Any objects that do not ensure this level of independence are considered Managed objects.

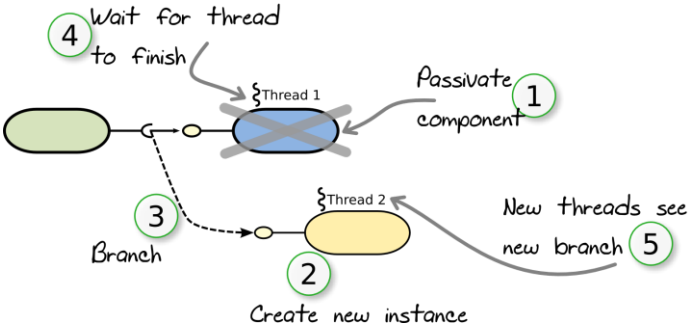

Figure 49: The proactive change of active components does not corrupt decoupled components because the component can be gracefully removed and active requests are allowed to finish. The time needed is not bound.

The amount of time needed to gracefully remove a component instance is unknown (*i.e.*, graceful removal is unbounded but finite[94]); however, graceful removal can fallback to a reactive (forceful) removal process, causing the requests to be aborted. After a configurable amount of time the framework aborts the request and forcefully stops and removes the component.

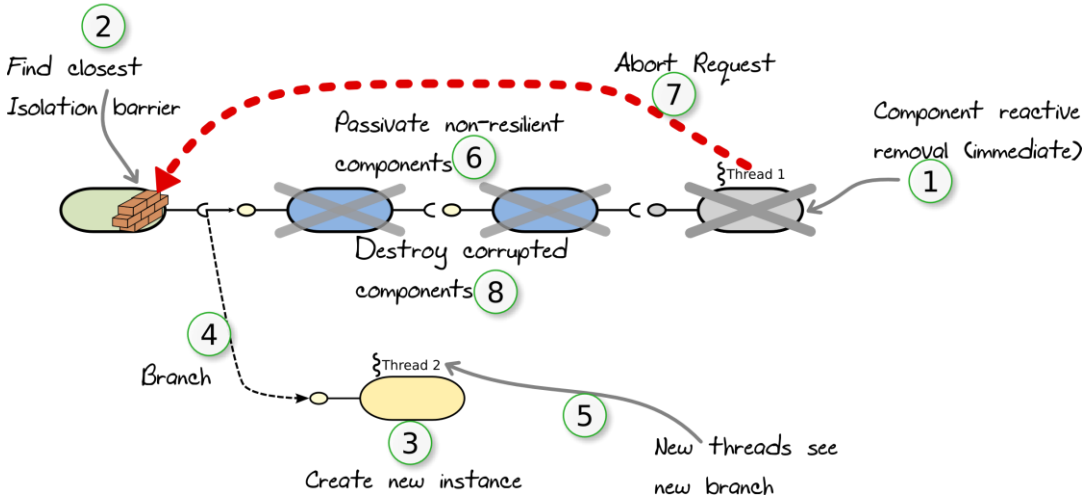

Figure 50: Reactive removal of an active component causes requests to be aborted. A `ServiceUnavailableException` *is thrown when aborting a request. Dynamically-resilient components remain consistent and valid because they can recover, while non-resilient components are potentially corrupted by the exception and will be removed.*

Reactive change, as for example a component's failure, causes the component to not be able to provide its service to others. If the component is active, any open requests on it are aborted. To abort a request we throw a `ServiceUnavailableException`, which can be caught by other components. If the calling component has a dynamically-resilient dependency, meaning that it catches the exception, manages it either internally or externally and remains consistent (see section 7.2.2), it is not destroyed. However, if the component's dependency is not dynamic-resilient, the component could potentially be corrupted and will be destroyed. The exception propagates to the next component where the same analysis is performed. Figure 50 shows this process.

[94] The underlying hypothesis that allows us to ensure that removal will eventually occur is that threads that a component receives must perform their task and then be released. Components cannot "steal threads" they did not create. Eventually, every thread is expected to finish. We consider stolen and dangling threads to be software bugs.

Sessional services require that the same component be used for multiple service interactions. Sessional services require the dependency not be changed until the sessional request has finished. In a proactive change, the component is held until the session is finished (*i.e.*, all threads finish the sessional request). A reactive change, similar to above, causes the sessional request to be aborted and the `ServiceUnavailableException` to be thrown. The difference is that, given the sessional nature of the service interactions, components in an open session are not necessarily active and won't catch the exception. To ensure consistency in case of aborted sessional requests, the components are notified the request has failed. This is shown in Figure 51. If the component does not implement the necessary callback method, it is potentially in an inconsistent state, either supposing the request succeeded or having only partially executed the sessional request, it cannot be trusted and will be removed.

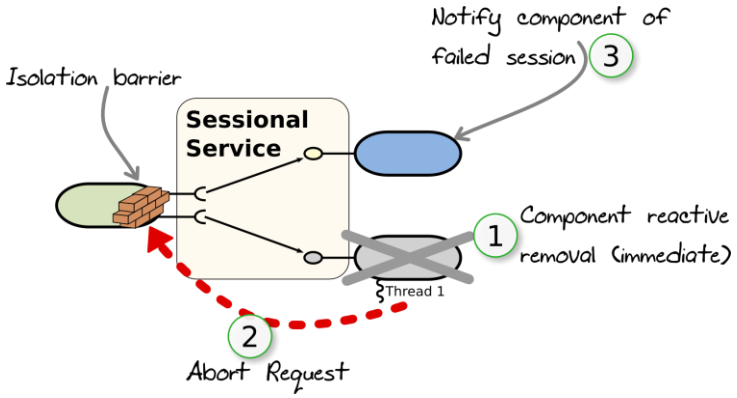

Figure 51: Aborting sessional requests causes the **ServiceUnavailableException** *to be thrown; however components that participate in the session must also be notified that the sessional request has failed. If they do not implement the callback for notifications then they are potentially inconsistent and are removed.*

7.3.3 Corruption analysis & application recovery

Dynamism can cause component corruption if the assumptions the component makes on its environment are incorrect. We have explained that the manner in which components are programmed directly influences the possibility that the component will be corrupted by dynamism. Developers and architects selectively choose which components and dependencies become resilient, allowing the framework to use this information to detect which components are unprotected and potentially corrupt. In the previous sections we have seen that, depending on each component's resilience to dynamism and its level of decoupling, the component can be considered consistent or potentially corrupt after a dynamic change event. At runtime, our approach is to automatically detect and remove any component instances that are suspected of corruption. This ensures consistency and proper execution at the cost of availability and state-loss[95].

Corruption analysis uses the different calculations of change impact from section 7.3.2 to determine the impact of a dynamic event on the entire application. As we have mentioned, an

[95] Removing components suspected of corruption can impact the application's uptime and can cause the state of the components to be lost. However, leaving components that are potentially corrupt may cause memory leaks or unexpected behavior at runtime.

application is considered consistent if all of its components are consistent. By definition, removing inconsistent components ensures consistency. Consequently, application recovery is the process of avoiding and removing inconsistencies, continuing execution using only consistent components. To recover requires deciding which components must be removed and where to branch the architecture for execution to continue (*i.e.*, which bindings to change to new components). We will use the example architecture in Figure 52 to illustrate the process of corruption analysis, branching and recovery. In the figure we can see that there are active requests, which are mapped onto execution threads. Moreover, the example shows coupled component instances (we use a lock on the dependency to indicate coupling, in this case component `d` is coupled to `e`), decoupled instances, which have dependencies that can be changed at runtime (*i.e.*, components `a, b, c`), and resilient dependencies (*i.e.*, component `a`) which can protect against the propagation of corruption from aborted requests.

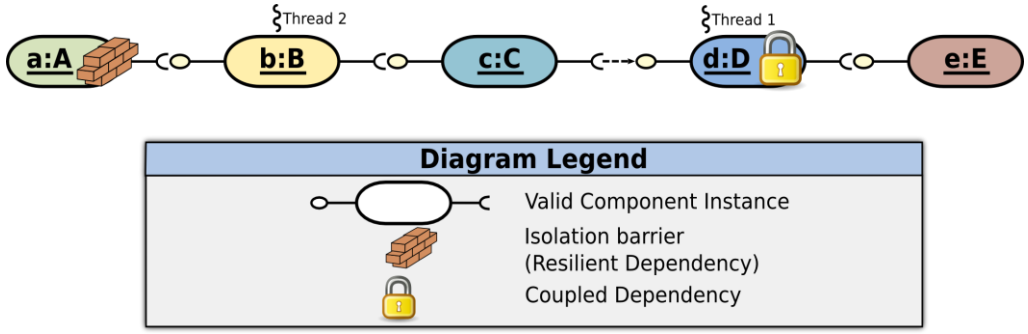

Figure 52: Example architecture showing components with active threads. Coupled components are shown, as well as components that implement isolation barriers.

Corruption analysis calculates what we call the *Corrupted Area*, which is the set of components that will be potentially corrupted after a dynamic change event. Corruption and recovery vary depending on the activeness of change. On the one hand, in a proactive change, none of the components are immediately corrupted, thus, the components can be passivated and gracefully removed from the architecture once they are no longer in use. A reactive change, on the other hand, immediately makes a component or series of components unusable. This can also cause open requests to be corrupted, further extending the list of corrupted components once the requests are aborted.

Naturally, the size of the corrupted area is dynamic and varies according to the current state of active requests in the application. The minimal corrupted area is the corrupted area calculated supposing no requests are aborted. The maximal corrupted area supposes that requests are aborted and propagate through all of the possible paths in the architecture starting from the point of initial corruption. Both minimal and maximal corrupted areas can be calculated statically at design-time. Interestingly, proactive changes always result in a minimal corrupted area[96] because no requests are aborted. This is useful because, ideally, our approach states that architects and developers identify volatile components, which result in reactive changes[97]; so if there is no volatility, dynamic changes should always result in a minimal corrupted area as long as the open

[96] This is true unless requests are forcefully aborted after a certain period. In such cases, proactive changes fallback to reactive changes.

[97] However, component failures also result in reactive changes and are much less predictable.

requests finish in a timely manner. Furthermore, because fewer components are corrupted, some state-loss is avoided and branching occurs closer to the point of failure. This is shown in Figure 53, where component `e` is to be removed. Once the change event has been identified, we prepare to invalidate all coupled components because their dependencies will no longer be valid (in this case it is component `d`). Outside of the corrupted (coupled) area, we branch all dependencies to valid components[98], in this case to component `x`, which can have its own distinct dependencies.

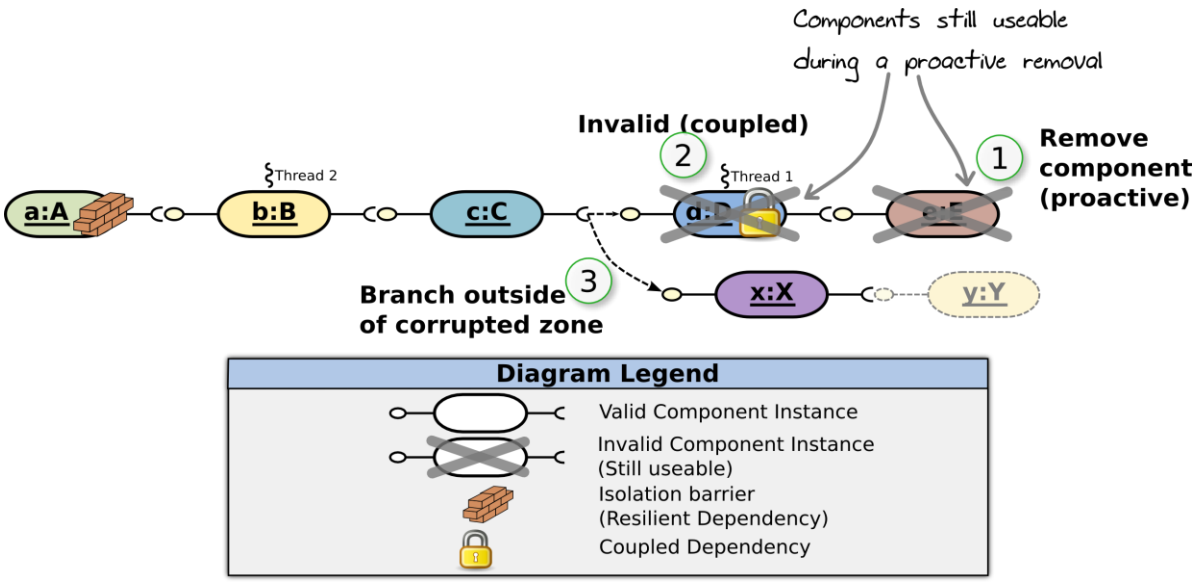

Figure 53: The proactive removal of a component.
(1) The removed component and (2) coupled components are identified. They are passivated and will be stopped once all requests have finished. (3) Branching at a decoupled dependency is performed; new threads will use this branch and will not see the old branch. The new branch is active before the old branch is removed, minimizing downtime.

Using the same example architecture, Figure 54 shows the calculated corrupted area given a reactive change (*e.g.*, a component failure). If the failure occurs while no requests are active, only coupled components must be removed and no exceptions are thrown nor requests aborted. Indeed, if the components aren't active, the calculation is the same as in the proactive example, resulting in the minimal corrupted area possible. If there is a request that must be aborted, then exceptions are thrown and all components in the request's execution path are corrupted until a resilient dependency is found. The figure shows that the request being executed by `Thread-1` is in the coupled area, which has been corrupted because of the failure of component `e`. The request cannot continue, it must be aborted. The `ServiceUnavailableException` exception is thrown and moves through the architecture (exceptions move up the stack, so, it crosses all components that have been partially executed by the thread and which were expecting the thread to finish). In the example, components `b` and `c` are potentially corrupted by this because they do not protect against the exception. Component `a` implements an isolation barrier, ensuring consistency and recovers the aborted request. From this safe point, we branch the architecture. All potentially corrupted components, denoted in the red area, are destroyed (*i.e.*, components `b, c, d, e`).

[98] Our work does not focus on selecting a component instance, this is left to the APAM framework. Robusta asks APAM to provide it with a valid instance that provides a service that is conform to the expected Service Contract. APAM may resort to deploying component implementations and instantiating in order to resolve a dependency.

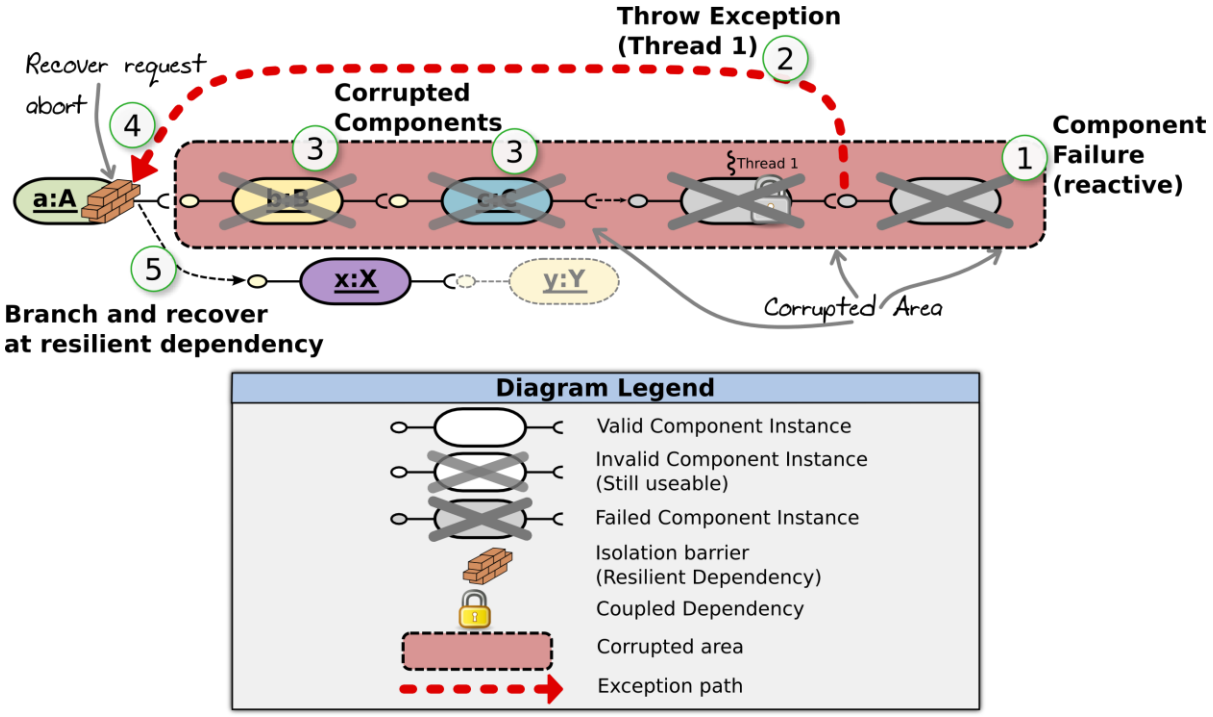

Figure 54: The reactive removal of active components.
(1) The failed component and coupled components are identified and stopped. (2) Exceptions are thrown that will cause (3) other components to be corrupted. (4) A dynamic-resilient component recovers the aborted request and (5) the framework branches from a consistent component.

As we can see, in the case of a reactive change there are different branching solutions depending on the current state of the requests. In order to minimize the impact at runtime of dynamism we should remove the absolute minimum number of components possible to ensure consistency. To do so, we will explore different branching techniques in the following section.

7.3.3.1 Reactive branching strategies

There are two branching strategies that we have informally described in the case of reactive changes. Optimistic branching expects the minimal corrupted area to result in a consistent architecture, and, attempts to branch at the closest dependency possible to the point of change. Pessimistic branching falls back to a safe branching strategy, only branching at dynamic-resilient dependencies that implement isolation barriers to ensure consistency. Nevertheless, using dynamic instrumentation, an optimal corrupted area can be calculated given the architectures current state and the state of active requests.

a) Optimistic branching

Optimistic branching supposes that the minimal corruption area will be sufficient to ensure consistency. During a reactive change, optimistic branching may fail because of active requests that attempt to execute failed component instances.

Figure 55 shows that the optimistic branching technique results in a branch that occurs immediately outside of the coupled area. This is the closest dependency that is properly decoupled

to allow rebinding. However, `Thread-1` may or may not require executing the failed component, which would lead to an inconsistent state and the request would have to be aborted. If there were no threads inside of the coupled area, then optimistic branching would be sufficient because all requests could be immediately sent to component `x`. It should also be noted that components coupled to a failed component are invalidated, but are not immediately corrupt, and can continue to service requests. Indeed, optimistic branching exploits this by allowing active requests to finish, just like in a proactive change, with the exception that there is a possibility that the branching strategy will fail (proactive changes do not result in failures or corruption).

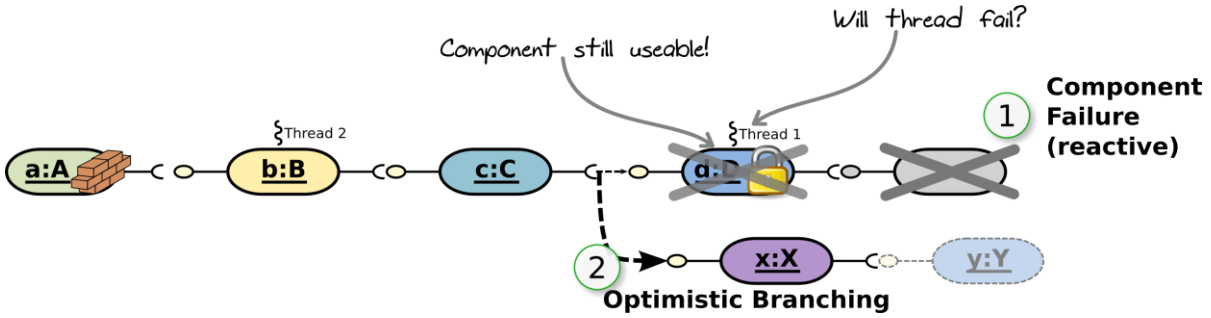

Figure 55: Optimistic branching strategy, showing that active threads are at risk of failing. It is unknown if Thread-1 will fail because it might require executing the failed component.

Depending on the architecture, optimistic branching may avoid corrupting and destroying components that have important state or that are expensive to construct. Furthermore, because branching can be parallelized, this eliminates the need to stop requests for long periods while the application is reconfigured.

b) Pessimistic branching

Pessimistic branching supposes that threads that might potentially fail, will fail. Pessimistic branching finds the nearest dependency that implements an isolation barrier and branches from there. This allows branching and preserving consistency no matter what state the active requests are in because the recovery mechanism implemented by the component with the isolation barrier ensures the aborted requests become consistent.

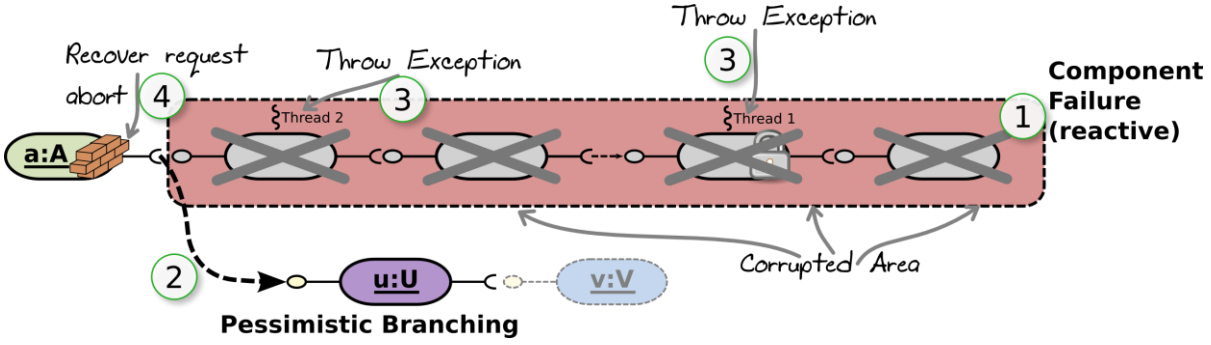

Figure 56: Shows pessimistic branching.
(1) A failure immediately causes coupled components to fail. (2) Branching is parallelized, should new requests come through component `a`, the architecture will be valid and service them. (3) All active requests are aborted, with the exceptions being caught and the requests recovered by components that implement isolation barriers.

Pessimistic branching is shown in Figure 56. Compared to the optimistic branching strategy, more components are stopped and all active requests in the corrupted area are aborted. There's no need to wait and see if the requests will require the failed components, everything is simply destroyed. Branching occurs where consistency can be ensured, that is, at dynamic-resilient dependencies.

c) Optimal branching

Optimal branching should result in the minimal impact on the application given any of the possibilities of failure and request abortion previously mentioned. However, because we do not know what execution paths an active request will actually take, it is not possible to pre-calculate the optimal branching strategy. Nevertheless, our approach is to initially attempt an optimistic branching strategy and then, in case of failure and aborted requests, fallback to a pessimistic branching strategy.

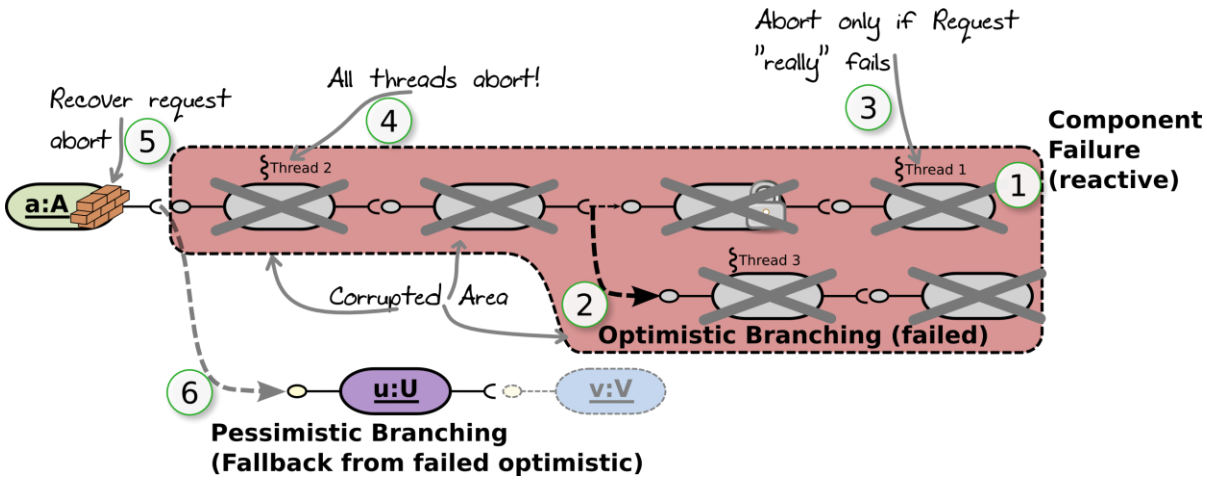

Figure 57: The optimistic branching strategy falls back to pessimistic branching.
(1) An initial reactive change resulted in the (2) calculation of a branching point. However, (3) Thread-1's execution path attempts to execute the failed component and must be aborted. (4) Thread-2 is consequently aborted also. (5)Tthe recovery mechanism ensures the request are recovered and (6) a new branch point is found. Finally, all corrupted components, including the failed optimistic branch, are destroyed.

Falling back to pessimistic branching is shown in Figure 57. The main defect of this approach is that, should the optimistic branch be corrupted and fail, the resources used to construct the branch will have been wasted. Indeed, and particularly in our example architecture, it might not be wise to attempt optimistic branching given that `Thread-1` has little flexibility in avoiding the corrupted component. Should, for example, component d have been connected to multiple instances and only one of them fail, we would have had a higher expectation that the optimistic branching strategy would be successful.

In order to increase the precision of our branching strategies, it is important for the framework to be able distinguish threads and their possibly different execution paths. The execution paths are important because aborted requests will corrupt components that have been partially executed and which await the thread to return. In order to determine which components are at risk of corruption, we have created a thread component stack.

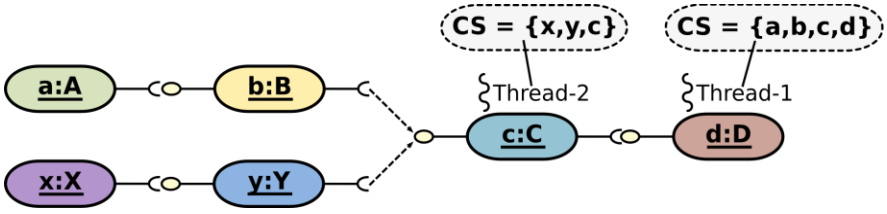

Figure 58: The thread component stack stores a record of components the thread has executed. This information is used to determine the optimal branching points in the architecture and to avoid destroying components when possible.

The thread component stack is shown in Figure 58. As we can see, each thread stores a list of components that it has partially executed but which have not finished. For example, `Thread-1` executed components `a, b, c,` and `d`. Should `Thread-1` be aborted, it is evident that components `x` and `y` would not be corrupted. The inverse is true for `Thread-2`, it would corrupt `x` and `y` but not `a` and `b`.

Finally, it should be noted that the thread component stack can be used for other purposes, such as request tracking, request-based QoS policies, debugging, and performance analysis. However, given that this is an optimization that comes at some cost at runtime it remains optional and is not required by the approach.

7.3.4 Summary of the Consistency & Recovery process

We summarize the process that ensures the application remains consistent and continues to operate correctly after dynamic change.

1. Calculate the impact of a change
    a. Removing a modules affects component implementations and component instances,
    b. Removing component instances can invalidate coupled component instances and requires branching at safe dependencies.
2. Calculate corrupted area
    a. Coupled components because of Managed objects (data corruption),
    b. Failed components and aborted threads (thread corruption).
3. Instantiate new components and branch at safe points
    a. Outside of corrupted area.
4. Gracefully stop corrupted area
    a. Passivate components,
    b. Hold exceptions until requests finish or timeout occurs.
5. Remove corrupted area
    a. Throw exceptions,
    b. Stop components,
    c. Cleanup architecture.

7.4 Conclusion

This chapter details how we protect components from failure and how they are rendered resilient to dynamism by means of isolation barriers and localized recovery mechanisms. We also detail how the mechanisms function at runtime to ensure consistency. These mechanisms and calculations are shared at design-time in order for architects to understand the expected dynamic behavior their applications will exhibit.

Given how dynamism can lead to corruption and corruption can propagate throughout the architecture, architects and developers need to have tools and underlying support to understand and contain dynamism. Building dynamic-resilient components, used for the construction of dynamic applications, is a difficult and error-prone process that can lead to unexpected or undesirable behavior if detailed attention is not paid.

Finally, the marriage between execution and design-time becomes apparent given the need for the framework to verify the architect's hypotheses regarding the application's expected dynamic behavior. Our approach uses the same algorithms at design-time that are used for calculating potential corruption at runtime, with the purpose of calculating the impact of potential change and improving the architect's understanding of the software. This support allows verifying that the application behaves properly (*i.e.*, as expected), while still ensuring consistency given expected or unexpected dynamism.

Building complex dynamic applications can be undertaken only if programmers and architects are supported by tools and frameworks dedicated to the analysis of dynamism in the application. Given dynamism's invasive nature, this support is required across the various levels of software design and execution.

Chapter 8

Architectural Support for Building Dynamic Applications

"The life of a software architect is a long and sometimes painful succession of suboptimal decisions made partly in the dark."
—Philippe Kruchten

"Unix was not designed to stop you from doing stupid things, because that would also stop you from doing clever things."
—Doug Gwyn

In this dissertation we have explored many of the complexities associated with the design and construction of dynamic applications. We have shown that dynamism cannot be entirely transparent and must be built into the application in order to ensure proper behavior. Furthermore, we have expressed the need for the runtime to ensure the application remains consistent no matter what dynamic event occurs. Consistency is considered more important than availability. Indeed, we build *expected dynamism* into the application's design and we handle *expected* and *unexpected dynamism* at runtime.

Given the difficulty of building dynamic components and of integrating them into a dynamic application, there is a need for tooling and assistance that guides architects and developers. Tools should assist in the comprehension of dynamic change in the application and to ensure the application behaves at runtime as the architect expects. This guides the architects to identify where dynamism is required and the levels of resilience the application needs. Tools also help developers to ensure their components meet the dynamic requirements architects have established. Ensuring a component is properly decoupled and resilient to change is essential to ensuring that the application will behave as the architect expects it to.

In this chapter we will explore the architectural support afforded to architects and developers in the design and construction of dynamic applications.

8.1 Building dynamic applications

Architects are naturally confronted with a tradeoff that occurs because of the sometimes conflicting needs of designing new components versus integrating existing components. Designing new components requires that the architect specify a component's dynamic requirements in order for developers to program the component accordingly. Integration and composition of components to build a dynamic application requires obtaining metadata regarding how the component was programmed in order to deduce how the component will behave in the current architecture. It is interesting to understand the differences between the approaches because the usage an architect makes of the calculations and tools we propose will be different.

Indeed, designing new components imposes dynamic requirements that the developers must follow, and verification and analysis rely on making sure the component conforms to its specification. Integrating existing components, on the other hand, allows developers to hypothesize on the application's desired dynamic behavior, while verification and analysis rely on verifying that the hypotheses are correct.

Moreover, many applications are a tradeoff between using existing components, code, libraries and utilities and integrating them into the application, and building new components. Figure 59 shows the cycle that exists between design and development. An architect designs the dynamic application, which feeds developers new and changed requirements that must be implemented. Components that have already been developed are composed and their existing dynamic behavior properties are analyzed, giving the architect feedback on how the application is expected to behave. As we can see, this cycle mixes characteristics of top-down and bottom-up development approaches.

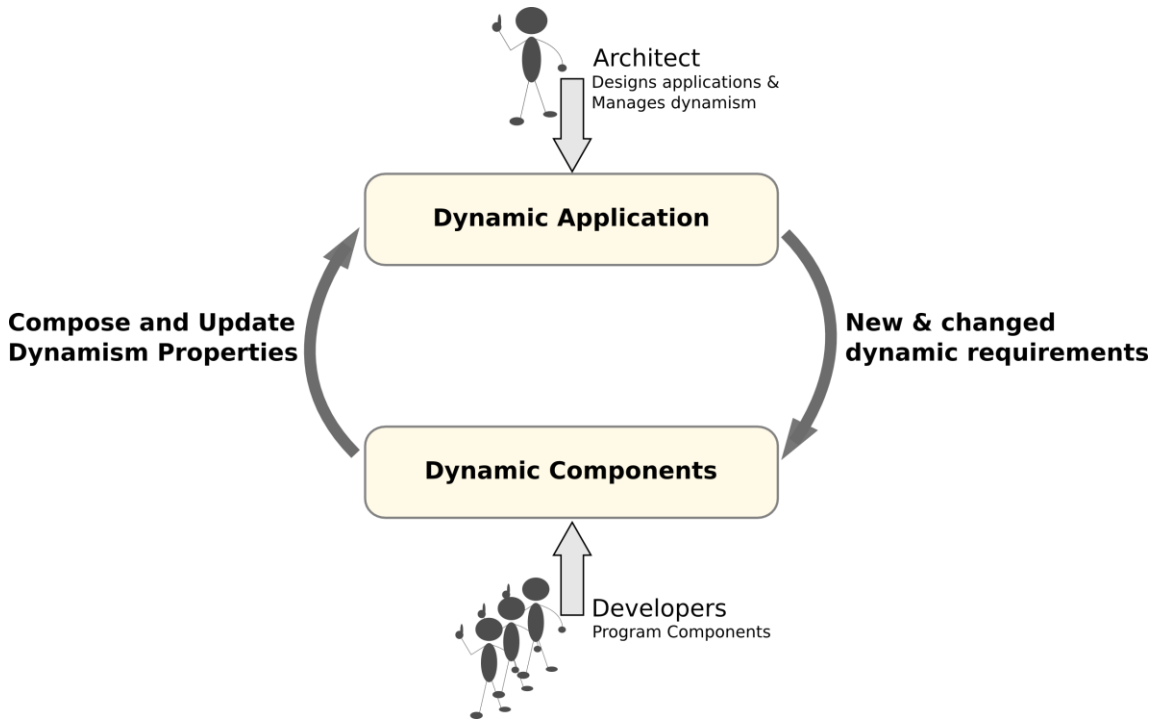

Figure 59: The development cycle in our approach to building dynamic applications.

Our approach allows refining components and the architecture in order to gradually achieve the dynamic behavior that is desired or acceptable. It also verifies expected dynamic behavior and improves the understanding of how corruption propagates by relatively simple dynamic changes. This permits the application to evolve, both at design time and at runtime. Indeed, dynamic software evolution is not only possible, but our approach actively strives for it. Informally, our approach promotes using runtime feedback in the ongoing design of dynamic applications. Figure 60 shows the relationship between design and execution, and how information obtained at runtime is incorporated into design decisions, resulting in updated versions of the application. This allows the application's dynamic behavior to be continuously refined.

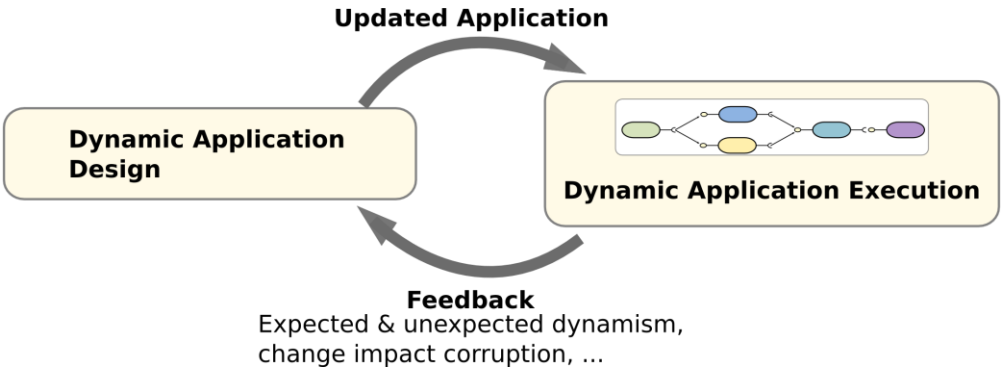

Figure 60: Feedback from execution leads to improved designs and refined dynamic behavior.

The use of feedback from runtime serves multiple purposes, as for example, verifying that design-time assumptions regarding dynamic behavior are correct and serve the application properly. Unexpected dynamism, as for example, component failures or forced updates to stable components, can then be incorporated into the design. Expected dynamism that is useful can be corroborated, and when not useful, it can be relaxed. These considerations serve to continuously place design and programming effort around components that require the levels of resilience and decoupling necessary to ensure consistency and an acceptable level of dynamism.

8.2 Architectural Analysis

The objective of analyzing the architecture of a dynamic application is to determine the components and areas of the application that are at risk of corruption caused by dynamism. We have characterized components according to their expected dynamic behavior, which can be *stable*, *detachable* or *volatile* (see section 5.1). This allows architects to selectively indicate which components are expected to change at runtime. Change can be intrinsic to the component, because it represents a physical device or network connection that can become unavailable at any moment, or it can be contextual, because the component is expected to be changed, removed or updated at runtime to make way for, for example, new features. Either case, architects are best placed to understand and anticipate the nature of a component at design-time. We call these assumptions, particularly regarding detachable and volatile components, the application's *expected dynamism*.

As described in section 5.3, we propose two calculations around a single concept that allow architects to better manage and understand expected dynamism. Namely, we use ***component zones***

to group components in order to *confine* or *protect* against dynamism. This allows the design of areas of the architecture that are ensured to not be corrupted by dynamism (*e.g.*, core components or the software's "backbone"), while allowing other areas of the architecture more freedom to change (*e.g.*, plugins). Component zones provide two calculations to manage dynamism:

1) if dynamic components exist in the zone, verify that dynamism is confined and does not propagate (we call it a **confinement zone**); and,

2) if dynamic components exist outside the zone, verify that the zone is protected from exterior dynamism (we call it a **resilient zone**).

When zoning the application, the architect is provided with information regarding which zones are corrupted by either exterior or interior dynamism, and through which dependencies the corruption occurs. Furthermore, component zones have their own dynamic behavior and can be stable, detachable or volatile, allowing the architect to establish the property that he expects and have it verified, or allowing the framework to indicate the calculated behavior of the component zone. We detail the types of component zones that architects can use and the dynamic behavior of component zones in the following sections. Finally, zone calculations can be enhanced to provide additional information and verifications.

8.2.1 Component zone types

The use of component zone calculations are beneficial in a top-down approach—designing components that must implement the defined dynamic behavior—or in a bottom-up approach—integrating components and verifying that the architecture satisfies the desired dynamic behavior. There are two primary zones: ***confinement zones*** ensure that expected dynamism does not propagate outside of the component zone; and ***resilient zones***, that ensure that exterior dynamism does not propagate to the interior of the component zone. Both calculations revolve around verifying that frontier components are contextually-resilient, that is, that no expected dynamism can corrupt the components, either from the interior or the exterior. It should be noted that component zone calculations use the change impact analysis described in section 7.2.

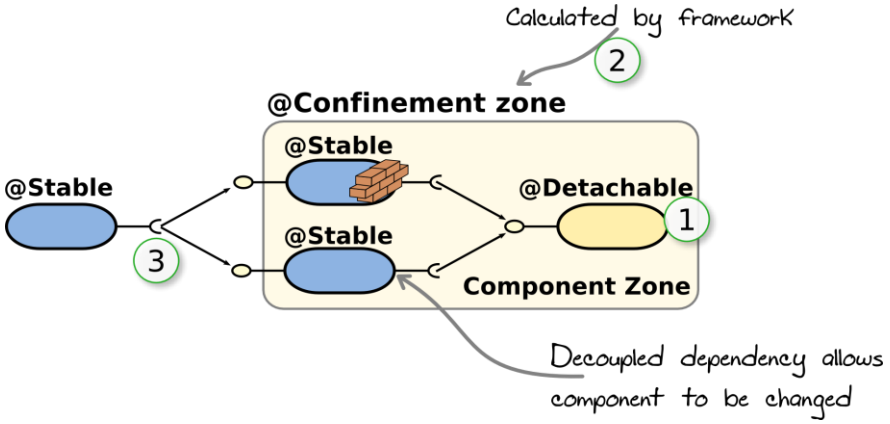

Figure 61: Example of the calculation of the confinement zone property.
(1) Starting from interior components that exhibit dynamic behavior, calculate potentially corrupted frontier components. (2) Calculate confinement property which depends on all frontier components being contextually-resilient. (3) Exterior components are not corrupted by expected dynamism originating in the component zone.

Figure 61 shows an example of a bottom up approach to calculate the confinement-zone property on a component zone. First, we must verify that dynamic components in the zone do not corrupt the frontier components. In this case, we see a detachable component that is guarded against by both frontier components (dynamic decoupling is a sufficient resilience to protect against detachable components). Because the frontier components are stable and contextually-resilient, the component zone properly confines dynamism. Exterior components are assured not to be corrupted.

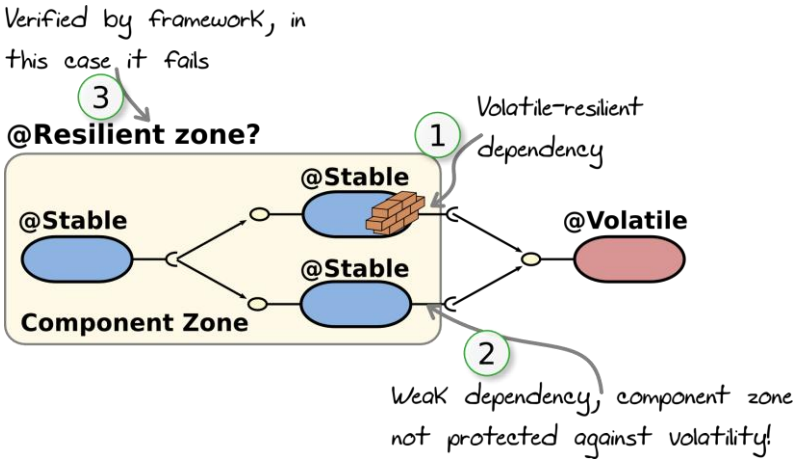

Figure 62: Example of the verification of the resilient-zone property. (1) The dependency is resilient to the volatile component, but (2) an insufficiently resilient dependency is found. (3) Because the component zone can be corrupted, the resilient-zone property fails.

Resilient component zones must protect the zone from dynamism that occurs outside the component zone. Figure 62 describes an example of a verified resilient-zone property (we use the question mark to describe that the architect has proposed the property and that it should be checked). As we can see in the example, the component zone fails the verification because a frontier component is insufficiently resilient. This allows the architect to either accept this condition and allow the component zone to be corrupted at runtime (corrupted components will be automatically removed by the framework but may result in state loss), or to increase the resilience of the weak dependency and send it back to the developers to implement the change.

We provide further verifications that an architect can use by combining or altering the existing component zone calculations. A component zone that is both a confinement and resilient zone is impervious to any expected dynamism in the application. We call such zones, ***dynamic-proof zones.*** A dynamic-proof zone ensures that dynamism that occurs inside the zone does not propagate to any exterior components, and conversely, dynamism from the exterior components does not propagate to the components inside. Additionally, a resilient zone that does not contain any dynamic components is both ***dynamic-proof*** and ***dynamic-free.*** As mentioned earlier, dynamic-free zones allow their interior components (not frontier components though) to be easily programmed without any concern for dynamism.

The use of dynamic-proof zones is fairly straightforward and is verified in the same manner as confinement and resilient zones. However, dynamic-free zones provide a couple of different characteristics that we would like to emphasize. To start with, a dynamic-free zone can only be

built using stable components. This is a simple verification; the architect cannot put dynamic components into a dynamic-free zone. An example is shown in Figure 63.

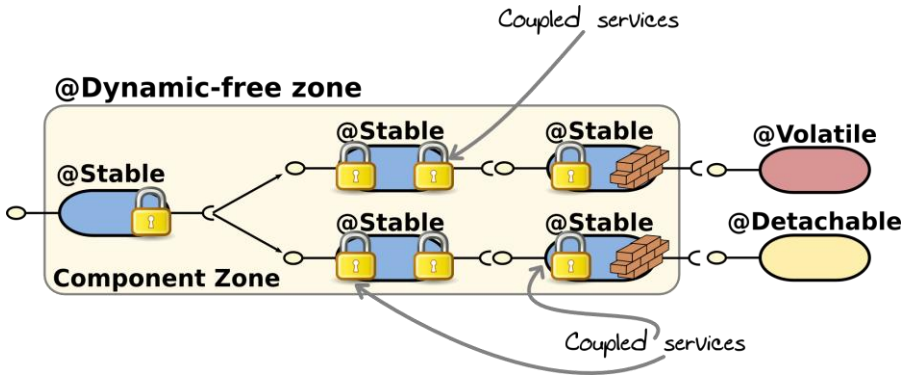

Figure 63: Example of a dynamic-free zone, where interior components can be coupled and require no particular dynamic programming restrictions, while frontier components need only protect from exterior dynamism.

By excluding dynamic components from the component zone, a dynamic-free zone allows components to be programmed without any dynamic programming restrictions. That is, components do not need to be dynamically decoupled (services and modules can be coupled, and managed objects can be shared) and no isolation boundaries need to be implemented either, except for frontier components, which still need to guard against exterior dynamism. This is particularly useful for well identified components that are crucial to the application, as for example, components that contain a lot of state that should not be lost, or core components.

Another use for dynamic-free zones, albeit indirect because our work does not focus on it, is for selecting components when resolving dependencies. Indeed, in the APAM framework architects can define partial architectures by leaving component dependencies unresolved. APAM then resolves the dependencies as-late-as-possible by selecting a component that meets the desired criteria. In the case of a dynamic-free zone, APAM selects components that are stable in order to not put the zone at risk of corruption. Thus, zone properties can be used for selection purposes and still ensure the application exhibits the expected dynamic behavior at runtime.

8.2.2 Dynamic behavior of component zones

Regarding dynamic behavior, component zones closely follow the same conditions previously established on components, with the exception that the zone must follow such restrictions as a group instead of an individual component. The calculated dynamic behavior of a component zone is the same as the calculated dynamic behavior of the "most dynamic" frontier component of the component zone. That is, if at least one frontier component is volatile, the component zone will be volatile[99]. If at least one frontier component is detachable (and none are volatile), the zone is detachable. If all frontier components are stable, the zone is stable. Unlike components[100], component zones are free to add component instances, remove them or substitute

[99] We should note that if the component zone's dynamic behavior is volatile then the component cannot be a confinement-zone because dynamism escapes the zone and affects external components.

[100] Components are generally not expected to add or remove services at runtime. Indeed, if a component is valid it is expected to provide the same services, and when invalid, its services become unavailable. This is not a strict condition,

them, but any changes to the component zone's frontier means that exterior components will be affected and can no longer make static assumptions about the zone. We present a more thorough definition for component zone dynamic behaviors:

- ***Dynamic component zones*** can change the type, number or quantity of services they provide or require at runtime. There are two types of dynamic components: detachable or volatile.
  - ***Detachable component zones*** can be updated, substituted or removed during the application's execution. Detachable component zones can be progressively passivated allowing components to gracefully stop in order to avoid corruption (*e.g.*, corrupting the current execution threads).
  - ***Volatile component zones*** can immediately and abruptly become unavailable, which can lead to the corruption of the current execution thread(s) if it is not protected against or recovered.
- ***Stable component zones*** do not exhibit dynamic behavior and do not affect surrounding components[101]. They are used for zones of the application that remain relatively static and under tight control from the architect, like the core or backbone of the software. Stable component zones must provide and require the same services as to avoid affecting exterior components, and they may never change their own lifecycle (*e.g.*, unexpectedly shutting down is not allowed). Interior components in a stable component zone may, nevertheless, be dynamic. Dynamism should not escape the component zone.

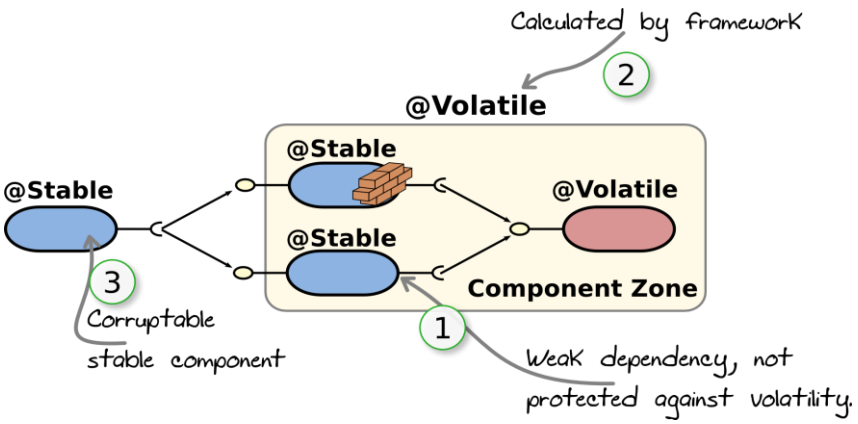

Figure 64: Example of a calculated volatile component zone.
(1) Volatility escapes the component zone through an insufficiently resilient dependency. (2) The component zone is calculated volatile, causing (3) exterior components to be determined insufficiently resilient and corruptible.

Similarly to component-zone types, component-zone behavior can be calculated or verified if the architect desires a particular behavior. Figure 64 shows an example component zone that is determined to be volatile because it does not guard against interior dynamism. In this case, the

but if a component changes its exterior aspect, i.e., its provided or required services dynamically, it must be considered either detachable or volatile and guarded against accordingly.

[101] Dynamic behavior is allowed in its interior.

component zone also fails the confinement-zone calculation of the previous section because dynamism escapes the zone.

At first sight it may seem redundant to calculate the behavior of a zone and to have the component zone calculations, such as confinement, resilience, dynamic-proof and dynamic-free. However, these properties are orthogonal and can be used in conjunction in order to better define the application's dynamic behavior. Figure 65 shows an example architecture where there is a core component zone that uses detachable plugin component zones. The plugins are used to add new features to the software at runtime and are expected to change, hence their detachable behavior. Nevertheless, the components of each plugin are programmed without dynamic restrictions, *i.e.*, in a dynamic-free zone, allowing them to make various static assumptions of their execution environment. Indeed, the architect can combine various dynamic behaviors and component zones to achieve more precise behaviors without losing the benefits of static verification.

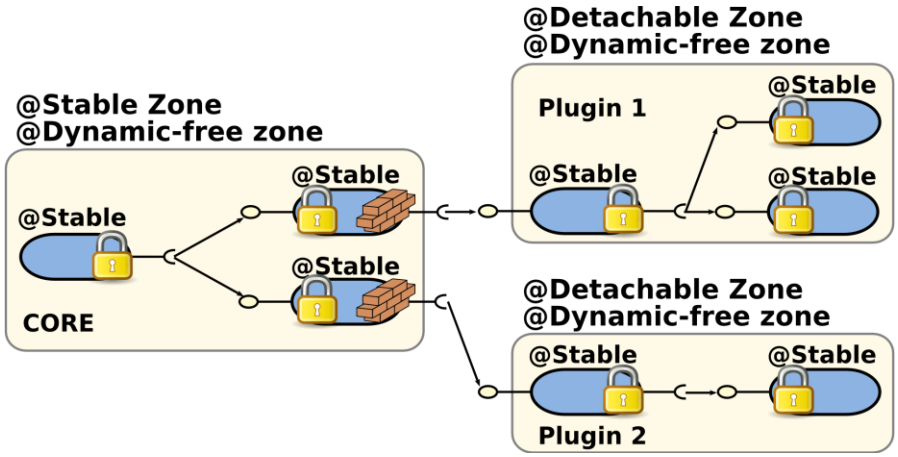

Figure 65: Combining dynamic behavior and component zone types allows for flexible architectures that allow defining desirable dynamic behavior while still verifying zone restrictions and programming constraints.

8.3 Component analysis

This dissertation has mostly focused on architectural concerns and on giving architects the power to decide where and how to use dynamism to meet the application's needs. However, once such decisions are made, the components must be implemented or adapted to meet those requirements. Indeed, when building dynamic applications, developers are in as much of a need for tooling and assistance as are the architects that design them.

This section provides an overview of the tooling and verification processes that are useful from the developer's point of view. Namely, we have studied the assistance that can be given to developers for decoupling implementations, for decoupling instances, for making dependencies resilient and for analyzing the propagation of coupling, such as propagating Managed objects and objects that are defined by hidden classes.

8.3.1 Decoupling implementations

Section 6.1 detailed the requirements for decoupling implementations. Namely, decoupling implementations requires identifying the Service Contract, starting from the service interface, and packaging classes into modules that can then evolve at runtime independently.

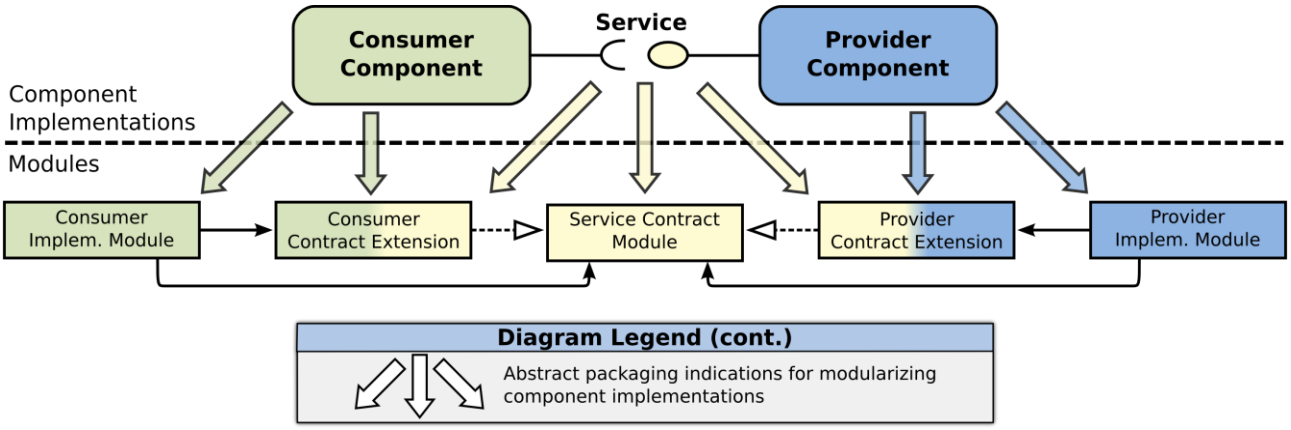

Figure 66: Summary of the 5 module approach for decoupling implementations.

Figure 66 graphically summarizes the approach to decoupling implementations by using a 5 module approach, in which the Service Contract and Contract Extensions are packaged into modules that are independent of the component implementation modules. In this way the implementation modules can be removed at runtime independently.

At design time, developers can be assisted thanks to the identification of the Service Contract. Once the service interface has been defined, we can calculate the transitive closure of classes that are referenced from the service interface, and propose packaging them into a Service Contract module or group of modules. Furthermore, a search for any classes that implement interfaces or inherit classes from the Service Contract is performed in order to place such classes into the Extension modules. All implementation classes can be left in their respective implementation modules. Tight cyclic dependencies between contract and implementation classes indicate that they are not easily separable and that the service should probably be redesigned. Finally, a verification process using the same analysis is possible to ensure that an existing packaging solution is indeed properly decoupled.

8.3.2 Decoupling instances

Service interfaces that use Managed objects must be specially handled in order to ensure the Managed objects are released when required. There are various calculations that can assist developers in ensuring that component instances are properly decoupled.

- Components that receive a managed object should implement the callback method for notifications to release the Managed object.
- References to Managed objects should not escape the component unless they do so through other Managed objects (see 8.3.4 Propagation analysis).

- Managed objects are probably better off being stored inside the implementation class's fields, and should avoid being copied throughout the component less a reference be incorrectly withheld. Developers can be warned of reference copies when compiling.

Finally, we have proposed the use of Free and Managed objects when defining a service interface. The difference in choosing one or the other resides in if the object can be used for as long as desired or if the object has an inherent retention policy that requires it being treated differently. We have alluded to the fact that Managed objects should be "special" objects, carefully and selectively used when necessary. However, from a practical point of view, when a service interface is initially identified, and before we know if the parameters and return values are either Free or Managed, it is unclear if they should be considered either Free or Managed by default. Indeed, if we were to err on the side of safety, any non-characterized object should be a Managed object, forcing developers to verify and indicate Free objects one by one. This would undoubtedly lead to lots of false coupling detections, where coupling is found although it does not exist. Doing the opposite, considering all objects to be Free unless indicated to be Managed could potentially lead to undetected coupling, causing memory leaks or unexpected behavior. Either solution requires configurability. In section 8.5 (Defensive programming techniques) we go over some best practices when implementing services. Such practices can also be detected and would assist in identifying Free objects.

8.3.3 Dependency resilience

We have described component and dependency resilience in section 7.2. Resilience can be internally handled by the component (*i.e.*, application specific isolation barrier) or externally managed by the framework. The former is easy to verify at design-time by checking that service invocations are wrapped in a `Try-Catch-Finally` clause that catches the `ServiceUnavailableException` and recovers. The latter is more difficult to detect automatically.

In essence, the resilience of a dependency depends on how corruptible the service invocations are, should the service fail at runtime (*e.g.*, a volatile component becomes unavailable). Because a service invocation passes parameters from one component to the other, we are specifically wondering how corruptible these parameters are. In the case of application specific isolation barriers, we expect the component to verify the parameters and recover or decide to fail. However, in the case of external recovery mechanisms, it is more difficult to know if the parameters are still valid and if we can re-invoke the service once we find another valid provider. By default, we consider the parameters to be invalid unless the developer tells us otherwise, meaning that by default components are not resilient. However, there are two calculations that can assist in automatically finding out if the invocation is safe.

The first solution is to verify that the receiving component does not modify or further propagate the parameters. If the component never changes the parameters then there is no reason for them to become invalid. If they are propagated, the same calculation can be performed on the next component to see if they are modified or propagated. The main problem with this calculation is that it can only be done once we know the architecture, which is not intrinsic to the component itself. It is a natural candidate for dynamic instrumentation with the objective of minimizing the impact of unexpected dynamism by improving corruption targeting.

The second solution is to analyze the corruptibility of the parameters themselves. Immutable or incorruptible parameters are naturally safe. These properties are further explored in section 8.5. If all parameters are incorruptible then we are sure the external recovery mechanism can be used.

The third solution is to see if the parameters are defensively copied, that is, that the invoking component makes a copy of the parameters before invoking the service (although, if this is done then the required step to implementing an application-specific recovery mechanism is slim), or the receiving component makes a copy before modifying the parameters.

These properties can be resumed as:

- @Unchanged: the component does not change the parameters.
- @Immutable: the parameters cannot be changed and are inherently safe.
- @DefensiveCopy: the receiving component makes a copy and does not change the originals (implies @Unchanged). Defensive copies can be shallow (only the initial object is duplicated), deep (the entire object graph is copied) or lazy (an initial shallow copy where a deep copy is performed on a write operation).

8.3.4 Propagation analysis

We have presented an initial case of coupling propagation by means of passing Managed objects in section 6.2.7 (Coupling propagation: passing Managed objects), and we have shown how a group of components can become unknowingly coupled. The same basic principles of propagation apply to passing other objects, such as passing objects defined by hidden classes (*i.e.*, classes not properly decoupled from implementations and put into the Extended Service Contract) and of passing parameters that we wish to verify have not been corrupted.

To assist developers, we can determine at design-time if a component actually receives, retains, uses or propagates an object that causes coupling. Indeed, components that are not programmed to be aware of dynamism—for example they do not implement notification callback methods—might in fact not actually retain the coupled objects.

If a component is potentially coupled because it interacts with a service that uses a Managed object or has a hidden coupled class, we can verify if the component retains the object or not. More specifically, we are interested in the following properties:

- @Stored: the component saves the reference in its internal state (*e.g.*, its fields). This indicates the component is effectively coupled.
- @Transient: the component only uses the object reference for the duration of a method but does not retain the object. This indicates that the component is not coupled.
- @Propagated: the component leaks the reference by passing it to other components. This means that it contaminates others. In the case of Managed objects and hidden classes, this means that other components are unknowingly coupled.

It should be noted that all of these calculations exist and are used for various reasons by other frameworks. For example, the Java Virtual Machine uses “escape analysis” for optimization

purposes in order to know if an object leaves the body of a method, *i.e.*, is stored, wrapped, passed on, which is basically the same information we are looking for too.

To summarize, in general a component that does not implement callback methods for notifications should not be passed Managed objects (unless this is a design consideration and, for example, the components are put into a Dynamic-Free zone). If the component does receive coupling objects, it should not propagate them. If they are propagated, we need to check to see how the next component handles them.

8.4 Static analysis versus dynamic analysis

Static analysis refers to analysis that is performed on source code or the architecture at design-time, when the application is not running. Dynamic analysis is performed at runtime when the application is executing. The advantage of static analysis is that it can be used at development time to help developers improve their code. It does not add overhead at run-time. Static analysis can be used to calculate potential coupling [Abdurazik 2007], and is in essence pessimistic, but it does not detect "real" coupling, and as such, is a conservative calculation of the worse-case scenario.

Dynamic analysis can be much more precise and can detect "real" coupling the moment it occurs (*e.g.*, the moment a component receives a Managed object). However, dynamic analysis adds overhead because the application must be instrumented in order for the detection points to determine when and what type of coupling occurs. It is hard to determine which objects are coupled and which are decoupled. To ensure consistency, by default, objects should be considered coupled.

Another improvement of dynamic analysis over static analysis is that it can be performed at the component instance level, which is a finer grain level of detection than performing calculations at the component implementation level, as is the case with static analysis. Indeed, the number of component instances is much more variable and not known before hand, making static analysis on component instances not really possible without sacrificing the flexibility provided from dynamic instantiation.

Static analysis is particularly interesting for detecting decoupling, potential Free objects, isolation barriers, and so forth. For example, if we detect a service object that is immutable, it is very likely that this is a Free object and can be treated as one. Such a determination has a very high chance of success, even though some simple objects, like strings, which are immutable[102], can have semantics added to them that introduce coupling, such as using a string to specify a file name through which two components indirectly interact. Indeed, these calculations are not guarantees of Free objects but can serve in assisting developers. In fact, guaranteeing anything in such cases comes down to an estimated guess, where there's a probability of guessing correctly or of failing. This adds to our intuition that what is required is assistance for developers to make the pertinent

[102] Strings in Java are immutable because once they are created there are no methods that allow changing the objects internal values. This differs from other types of immutability, such as bitwise immutability.

decisions and understand their consequences, and finally it is the developers who, by means of additional metadata (*e.g.*, annotations) tell the framework what they did.

8.4.1 Combining static and dynamic analysis

Interestingly, and particularly in the case of propagation, static and dynamic analyses can be combined to allow for low-overhead fine-grained coupling propagation detection. Static analysis can provide information regarding the potential propagation of Managed objects, such that, we can see propagation paths through which the objects are shared throughout the architecture. Once the precise methods, parameters and return values that cause coupling and propagation have been determined, dynamic instrumentation can be used for those precise points, avoiding instrumenting the entire application. In this case, the framework can be aware of the exact moment components become coupled, without having a large runtime overhead.

Furthermore, dynamic instrumentation can also allow us to know if a coupled object has been released or not. Indeed, similar work on stale service references and dangling references [Gama and Donsez 2008] has been successfully implemented.

In short, static and dynamic analyses allow finding potential hazards to dynamism before and during execution. They also allow being more fine-grain when deciding if a component is potentially corrupt or not. Indeed, dynamic instrumentation can be very precise in determining that a component is not corrupted. In the next section we will present defensive programming techniques—often considered best practices—which assist dynamic applications by minimizing the possibility of coupling and corruption even further.

8.5 Defensive programming techniques

Dynamism opens up the possibility that, unexpectedly, bad things happen. Indeed, it is impossible to foresee all of the changes that will occur, making it quite relevant to attempt to curtail some problems as early as possible. Defensive programming intends to ensure that the software continues to run despite being used in unforeseen ways. Indirectly, defensive programming increases the general quality of software by reducing potential bugs. Furthermore, it tends to make software easier to follow and more readable. In dynamic environments, defensive programming can be particularly interesting because components, by supposing the components they communicate with will not properly handle dynamism, can proactively protect themselves from such situations. In particular, we are interested in the aspects of defensive programming that minimize the use of shared mutable state which complicate the development of multi-threaded and dynamic applications.

Not coincidently, Free objects also minimize the use of shared mutable state. Indeed, the use of Free objects is a defensive programming technique. Furthermore, this chapter has taken a look at various properties that characterize components and how they have been programmed. These properties are relatively un-intrusive and serve two purposes: first, they document the code and make it easier to follow and understand (and ensure invariants are not broken by developers in future versions); and second, they allow this information to become available at both design-time and runtime for corruption analysis.

Tools such as Findbugs[103] already permit various properties to be checked and or automatically calculated at development time in the Java language. However, Java is not a language that allows strong guarantees (*e.g.*, immutability) so these tools are not perfect. Nevertheless, this ensures that the annotations truly represent what the code does and verifies the properties are true.

Of the defensive techniques that would assist in creating dynamic applications, we can mention the following:

- Making all service objects immutable
  - This way they are never corrupted,
  - If strict immutability is not possible, at least make sure that they are not corruptible;
- For mutable service objects
  - Make defensive copies to quickly recover from unexpected failures[104];
- Always use isolation barriers
  - Wrap all service invocations in `Try-Catch-Finally` clauses and catch the `ServiceUnavailableException;`
- Never communicate through hidden mechanisms
  - *E.g.*, avoid coupling through shared files,
  - Make communication between components occur through the service and not second or third level shared objects obtained through an initial invocation;
- Don't leak service objects from other components
  - Unless they are free and immutable themselves,
  - Avoid sharing mutable components among various components;
- Attempt building stateless components
  - Store state in safe backends or push state towards clients,
  - If not, make components incorruptible;
- Make all fields private unless they need greater visibility;
- Make all fields final unless they require mutability
  - Attention should be paid to container objects because it is the reference to the container that is immutable, not the container's contents.

[103] http://findbugs.sourceforge.net/

[104] It is worth noting that the Java Virtual Machine performs many performance enhancements, such that, should the defensive copy not be necessary it is not performed internally. In essence, this allows defensive programming techniques to be free adding, zero overhead when not needed. http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html

Because of its importance, we recall the conditions required for achieving immutability. A class is immutable in Java if:

- its state cannot be modified after construction;
- all its fields are final; and
- it is properly constructed (the `this` reference does not escape during construction).

Immutable objects can still use mutable objects internally to manage their state but they must not change them or share references to them (*i.e.*, no leaking references). It is interesting to note that immutable objects (and hence Free objects) offer additional performance advantages such as reduced need for locking or defensive copies and reduced impact on generational garbage collection.

Additionally, and although not directly related to dynamism, Bloch [Bloch 2008] recommends using the following properties to document code:

- @GuardedBy: The field or method to which this annotation is applied can only be accessed when holding a particular lock, which may be a built-in (synchronization) lock, or may be an explicit `java.util.concurrent.Lock`.
- @NotThreadSafe: The class to which this annotation is applied is not thread-safe.
- @ThreadSafe: The class to which this annotation is applied is thread-safe.

In general, following these techniques can greatly improve performance, reduce bugs, and improve dynamism. They help decouple components by making shared state safe and immutable, adding robustness to the application.

8.6 Conclusion

Throughout this dissertation we have argued that managing dynamism is a difficult, cross-cutting task that is complex and error-prone. Improper handling of dynamism leads to unexpected and undesirable behavior, such as inconsistencies, corruption, memory leaks, among other problems. This chapter has provided an overview of our approach, from design to runtime, and back. We have described the types of analysis that can be performed at both the architectural level, as well as at the component implementation level, to assist architects and developers respectively in their quest to build dynamic applications.

Given the difficulty, invasiveness and cross-cuttingness that dynamism has on software, tooling and support for managing dynamism is required for both architects and developers. Developing dynamic applications requires assistance, such as our efforts to provide analysis, and guarantees, such as our approach to decoupling and resilience that ensures the application remains consistent given both expected and unexpected dynamism.

Chapter 9

Implementation and Validation

"Where you give software developers a choice of doing the simple thing or the more complicated thing, they go for the more complicated thing, because there's more reward for doing it."
—Hasso Plattner, Chairman SAP, interview with the Wall Street Journal on May 15th 2007

"You cannot control who you do not understand."
—Mao

This chapter presents the implementation prototype for Robusta and its validation. We have focused our implementation on a proof-of-concept prototype that demonstrates the feasibility of using and implementing our approach in large and complex software used in industry. We have particularly focused on the runtime aspects needed for Robusta to be a feasible and useful approach. Design-time aspects and tooling to assist in the development of dynamic architectures have not been implemented, but we know, by experience, they can be implemented. It should be mentioned that we have made an effort to follow an Open-World assumption in our prototype in order to account for the fact that we cannot anticipate the dynamic changes an application will undergo or what components, classes or modules will be used or changed in the future. Following such an assumption, our prototype performs its analyses as-late-as-possible in order for them to assess the current state of the application at any given moment. We have verified that such an approach can be used in industrial software to assert large-scale systems.

Our prototype is primarily focused on the detection of component and module coupling, as described in Chapter 6 Dynamic-Decoupling, which is the basis to permitting unexpected dynamism in an application. Ensuring a component is properly decoupled and resilient to change is essential to assessing that the application will behave as the architect expects. In order to achieve decoupling, our analysis focus on analyzing classes, detecting coupling, calculating the Service Contract and determining the Contract Extensions.

9.1 Requirements for coupling detection

As described in Chapter 6, in order to calculate the full extent of coupling in an application we must have complete knowledge over all the relationships between classes and interfaces that have been loaded into the application. As described, for each class and interface we must discover the *extends* and *depends* relationships. We represent the relationships among classes and interfaces

in a directed graph called the Class Dependency Graph. This graph contains all loaded types and their relationships in order to properly calculate the extent of coupling among types. This graph is computed at runtime.

Coupling occurs at the class-level but deployment involves modules that contain sets of classes, therefore we must be able to detect the relationship between classes and modules in order to determine which classes belong to which module. Classes are contained in modules and modules are the unit of dynamism for adding and removing classes at runtime (and hence functionality). However, as will be seen later in this chapter, it is not always evident to obtain this information. Furthermore, in order to follow through with current development practices, such an approach should account for the loading of multiple versions of a class or class name clashes for classes contained in different modules. Indeed, it is common for large applications to contain multiple versions of a same library that are used independently in different areas of the application.

In order to detect coupling we must calculate the Service Contract. Given the Class Dependency Graph, it is straightforward to calculate the Service Contract using simple reachability heuristics. Furthermore, the Contract Extensions need to be determined also. Service Contract Extensions can be particularly problematic at runtime because a single class can come along and extend another, causing hidden coupling that can be difficult track and detect. Such hidden couplings, as described in section 6.1.2, can cause undesirable and unexpected behavior because of their contamination of the Service Contract. This is interesting because this shows that attention needs to be paid to what is added to the application, not only what is removed, if we should keep things well decoupled and minimize the impact of dynamism on the running application.

The prototype must calculate the impact on the application when performing a specific dynamic change. In particular, previous to removing a module, it is necessary to calculate which modules will be impacted by the removal and would also require being removed, in a domino effect. We should note that at the architectural level, the Service Contract is expected to be independent from the component implementations in such a way that component implementations may evolve independently. Of course, these are design decisions but the prototype must allow for their verification.

Finally, an important requirement that has influenced much of our prototype and validation is the Open-World assumption we have decided to follow in order to approach as closest as possible real-life concerns that exist for modern and complex long-running modular applications. We use the open-world assumption to indicate that no single or central entity has the wisdom to foresee the dynamic changes that will occur in the future. This is essential to allowing programmers and architects the freedom to adapt their applications without having to predict each and every adaption in advance. Furthermore, in a more practical sense, it is unwise to expect that the runtime anticipate everything that is going to be loaded and thus perform all coupling calculations beforehand. Indeed, the open-world assumption follows the use of current industry technologies for building large and complex software, such as Java enterprise applications.

In summary, our prototype must achieve the following:

- Build a Class Dependency Graph containing all classes loaded and accessed by an application.
- Calculate the Service Contract and Extended Service Contract.
- Calculate change impact in order to, firstly, understand beforehand the interactions and impact of a change and, secondly, properly refresh all dependencies.
- Follow our Open-World assumption to allow for unanticipated and unexpected dynamism.

9.2 Solution Comparison and Tradeoffs

The assumptions we have made regarding how dynamic applications are developed and executed have influenced the technical decisions we have made both regarding our implementation and our validation. In this section we will analyze our choices regarding the implementation of our prototype.

9.2.1 Design-time versus runtime analysis

When to perform an analysis is important to determining the applicability of the solution. Given our important open-world assumption, we can quickly see that it is necessary to provide the analyses and calculations at runtime because we cannot anticipate what classes will be run nor the dependencies that will exist at design-time. Performing such calculations at design-time reduces the scope of Robusta's usability.

This is not to say that the analyses are not useful at design-time. Quite the contrary, they can be very useful in assisting developers to properly decouple their components at an early stage, avoiding the cost and energy spent in refactoring code late in the development process. Nevertheless, only at runtime can we have a complete picture of the target application and all the necessary data about coupling among classes and modules as it exists at any given point in time. Performing the calculations at runtime is widely applicable to different use-cases.

9.2.2 Bytecode versus source code analysis

The approach requires reading classes and calculating the relationships among them. There are two ways of doing this, either by reading human-readable source code or by performing compiled bytecode analysis. Following current programming techniques and the technologies used (e.g., Maven, Gradle, OSGi, Java EE, Grails), the execution framework rarely has access to all the source code used to compile or run a program. Indeed, the proliferation of libraries spread across an organization or obtained through third parties over the internet make it more and more common to simply recover existing compiled packages, often open source, and directly integrate them within an application.

This implication means that a source code analysis is insufficient to satisfy our open-world assumption, which requires a bytecode analysis tool. Again, this does not mean that source code analysis would be useless; on the contrary, sourcecode analysis is often a higher-level analysis that can assist development but is insufficient for the general case.

9.2.3 Automated analysis versus interactive diagnostics

Our initial prototype has focused on being an interactive diagnostics tool to calculate Service Contracts and to perform coupling analysis among classes and modules. We believe that at the current stage using this tool much like a debugger would be highly beneficial for the construction of dynamic applications. Indeed, the tool could be used to verify and weed-out any potential issues regarding dynamism. It also can be used during trouble-shooting sessions.

Ideally, such a tool would automate many procedures and provide insight into both choosing dependencies (e.g., dependency resolution techniques) and performing reconfigurations or minimizing change impact. However, such a tool is much more complex and requires a deeper understanding of the application and the desired dynamic behavior an architect may have specified. It also requires large heuristics that have not been discovered yet. In addition, existing guidelines never reached a consensus. It is also probable that a complex tool like that would require extensive configuration and per-application heuristics in order to be useful. As a consequence, we have chosen to address fully automated and autonomic tasks in future versions of our tool. Nevertheless, the underlying calculations and the basic tooling that we have implemented should serve as a base for such work.

In short, our prototype must follow meet these requirements:

- Analysis performed at runtime.
- Analysis performed on compiled bytecode.
- Interactive “debugger-like” environment.

A solution that performs runtime bytecode analysis is effectively the more general solution but arguably also the most complicated case to implement. However, these decisions follow our open-world assumption and should demonstrate the feasibility of our approach.

9.3 Implementation technologies

In order to implement Robusta we have extensively used multiple Java based technologies that we describe in this section. We have chosen Java for our implementation because it is extensively used in both open-source and proprietary projects, in both academia and industrial settings. It also provides a great amount of tooling and highly tested frameworks. An interesting distinction we should point to is that we are interested in Java the framework and virtual machine, not Java the language. As such, our work on dynamism potentially touches many different languages that execute on the JVM.

a) Java agent

A Java agent is a special piece of code, loaded in the Java Virtual Machine. This piece of code has access to low-level mechanism of the execution environment. More particularly, Java agents have instrumentation abilities that allow redefining the content of the classes loaded at runtime. Agents also allow recovering information that the Java Virtual Machine hides to regular programs. Agents must be specified at startup and are inherently static. They cannot be dynamically added or removed to the JVM at runtime in production[105]. In addition, they may introduce security threats, so must be used carefully. However, using an agent is not rare today. Debuggers and profilers are based on Java Agents. JRebel, the most well-known hot-reloading framework is based on a complex Java agent. The Crash shell is another example also based on a Java agent.

Java agents are executed within the Java Virtual Machine and not on top of it as a regular Java program. In addition, to implement a set of pre-defined methods (called by the Java Virtual Machine), Java Agents are packaged in Jar file specifying a set of specific Manifest entries:

- *Premain-Class*: specifies the main entry point of the agent. That is, the class containing the *premain* method. When an agent is specified at JVM launch time this attribute is required.
- *Agent-Class*: If the agent is attached after the VM has started then this attribute specifies the agent class. That is, the class containing the *agentmain* method. This attribute is mandatory; if it is not present the agent will not be started.
- *Boot-Class-Path*: Specifies the *classpath* of the agent. These entries are searched by the bootstrap class loader after the platform specific mechanisms of locating a class have failed.
- *Can-Redefine-Classes*: Enables or disables the ability to redefine classes.
- *Can-Retransform-Classes*: Enables or disables the ability to retransform classes.
- *Can-Set-Native-Method-Prefix*: Enables or disables the ability to set native method prefix.

According to theses entries, agents can:

- Intercept all loaded classes by the Java Virtual Machine
- Transform loaded classes, i.e., change their content before they are loaded
- Redefine classes, i.e., change their content after they were loaded.
- Modify the native method resolution

When an agent is attached to the Java Virtual Machine, the execution environment loads the agent main class, and calls the *premain* method (or the *agentmain* method for agents loaded after the JVM startup). From these methods, the agent has access to an *Instrumentation* object letting the agent to register *Transformers*. Transformers are provided by the agents and are called whenever the Java Virtual Machine defines a class. Class definition happens just before the actual loading of the class. At that step the content of the class can still be updated, as it's still *raw* bytecode.

[105] Java 6 has introduced the Attach API to load agents dynamically. This feature used by debuggers and profilers is disabled in production.

Java agent gives low-level access to the JVM's internals and should be used carefully, particularly in production environments.

b) ASM

The ASM library is a project of the OW2 consortium. It provides an API for reading, modifying, and writing bytecode. It can be used to modify existing classes or dynamically generate classes, directly in binary form. ASM provides common transformations and analysis algorithms allow to easily assemble custom complex transformations and code analysis tools.

ASM rely on a visitor pattern, where each adapter is called during the class visit. For instance, in the following snippet, a class reader reads a class given as a byte array. When the reader visits an element of the class (e.g., fields, methods, instructions), it delegates the visiting events to the first adapter of the chain (*cv*), which delegates to the next adapter. Here the called adapter is *cw*, a writer collecting the resulting bytecode.

```
byte[] b1 = ...;
ClassWriter cw = new ClassWriter(0);
ClassVisitor cv = new ClassVisitor(ASM4, cw) { };
ClassReader cr = new ClassReader(b1);
cr.accept(cv, 0);
byte[] b2 = cw.toByteArray(); // b2 represents the same class as b1
```

ASM is powerful and can be used to implement complex class transformations. The Apache Felix iPOJO component model is based on ASM. However its use is far from being trivial; developing complex transformations can be challenging.

c) OSGi: Isolation through classloaders

The OSGi™ platform is a Java-centric, centralized, service platform specified by the OSGi Alliance. Initially, the specification focused on residential and industrial gateways. At that time, the specification defined only deployment abilities and common services. With the fourth version of the specification, OSGi increased in popularity. In this version, OSGi defined how to build, deploy, and manage sophisticated modular applications. Nowadays, OSGi is widely used in application servers, large-applications, mobile phones and 24/7 gateways.

OSGi applications are packages in *bundles,* which are special Jar files specifying a set of metadata. These metadata instruct the OSGi runtime on how classes from the bundle must be loaded and how they access their class dependencies. Thus, bundles have the ability to import and export Java packages as well as declare dependencies on other bundles. The OSGi specification defines how these bundles are installed, resolved, activated, updated and uninstalled at runtime, and this without restarting the underlying OSGi runtime and Java Virtual Machine. The OSGi framework automatically resolves the dependencies between bundles listed above, but bundles can also publish and use *services.* Services are specified functionalities. Service bindings are not managed by the OSGi framework, and thus developers are in charge of them. Services are by

nature dynamic, they appear and disappear at any time. Managing this dynamism is pretty challenging and requires a high expertise in OSGi and in Java's concurrency model.

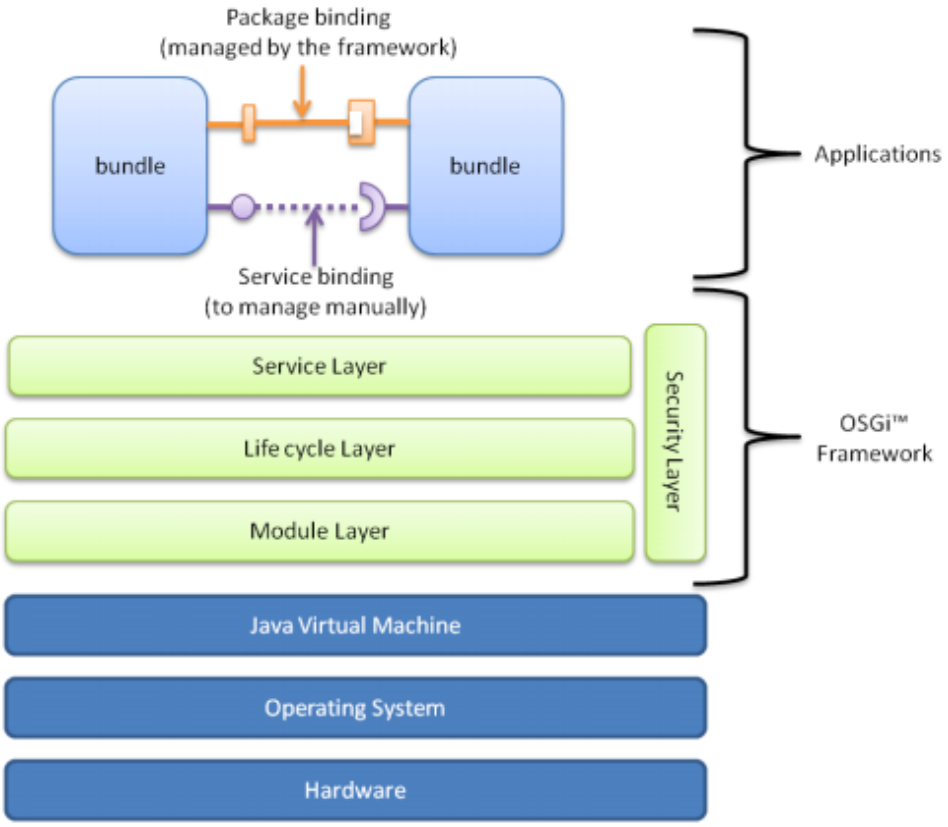

Figure 67: An overview of the OSGi runtime.

d) iPOJO

iPOJO is a service component runtime aiming to simplify OSGi application development. iPOJO is heavily used in academic and industrial projects. It natively supports all the dynamism of OSGi. iPOJO is designed to run modern applications that exhibit modularity and require runtime adaptation and autonomic behavior.

iPOJO aims to simplify service-oriented programming on OSGi frameworks; the name iPOJO is an abbreviation for injected POJO. iPOJO provides a new way to develop OSGi service components, simplifying service component implementation by transparently managing the dynamics of the environment as well as other non-functional requirements. The iPOJO framework allows developers to more clearly separate functional code (i.e., POJOs) from the non-functional code (i.e., dependency management, service provision, configuration, etc.). At runtime, iPOJO combines the functional and non-functional aspects. To achieve this, iPOJO provides a simple and extensible service component model based on POJOs.

An iPOJO service component is able to provide and/or require services, where a service is an object that implements a given Java interface. In addition, iPOJO introduces a callback concept to notify a component about various state changes.

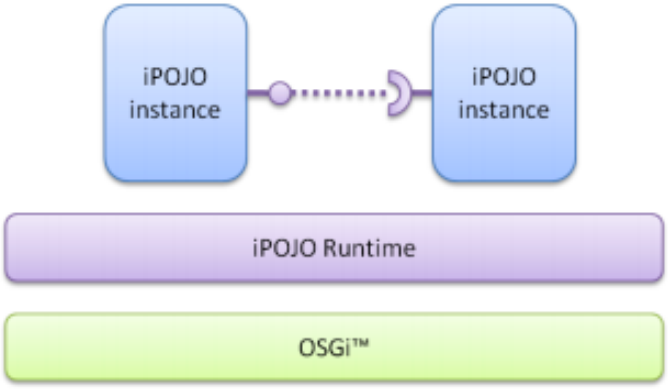

Figure 68: the iPOJO runtime.

The component is a central concept in iPOJO. In the core iPOJO model, a component describes service dependencies, provided services, and callbacks; this information is recorded in the component's metadata. After components, the next most important concept in iPOJO is the component instance. A component instance is a special version of a component. By merging component metadata and instance configuration, the iPOJO runtime is able to discover and inject required services, publish provided services, and manage the component's life cycle.

The iPOJO component model is extensible. iPOJO instance container is composed of a set of handlers, as can be seen in Figure 69. Even core features are developed as handlers. iPOJO lets developers provide their own piece of the container, i.e., develop their own handlers to manage unsupported concerns.

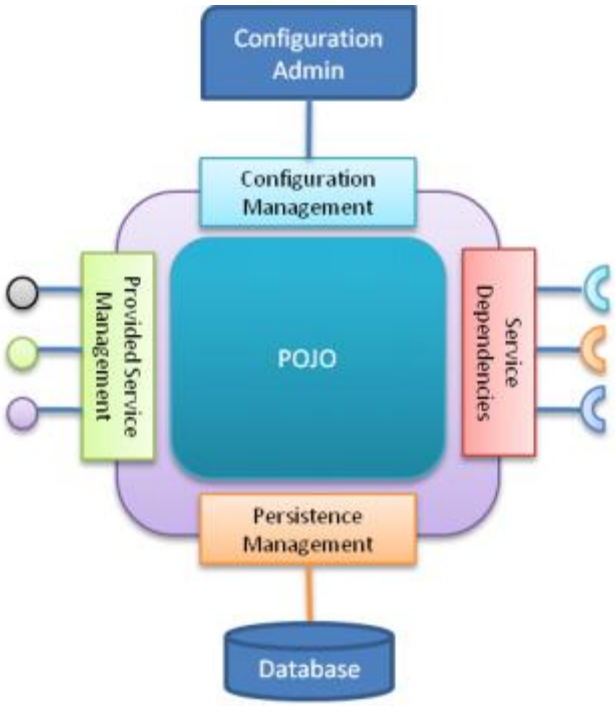

Figure 69: An iPOJO component showing its handlers.

9.4 Implementation overview

The Robusta prototype consists of three main elements: the Robusta Agent, the Robusta Bytecode Manipulator and the Robusta Application Analyzer. We explain them in this section.

9.4.1 Robusta Java Agent

The Robusta Java Agent is used for two purposes: the first, to obtain otherwise hidden information from the JVM, such as, the list of all classes that have been loaded up to that point; and the second, to transform classes with necessary metadata using the Bytecode Manipulator so Robusta can later recover the metadata and create the complete class dependency graph.

The Robusta Agent provides a hook that allows the Robusta Analyzer to obtain the list of classes that have been loaded by the JVM. This hook can be called numerous times and each time it provides an up-to-date list of classes. The agent obtains the list in a straightforward manner using the Instrumentation API provided by the JVM. However, it should be noted that the JVM provides an array of classes (specifically an array of *Class* objects) and not a proper dependency graph. The array provides each and every class loaded by the JVM, but a Class object does not allow for recovering the necessary dependency information in any easily accessible manner.

The second function of the agent is to add easily accessible dependency metadata to all classes that are loaded by the JVM. To do this, the agent subscribes to the JVM's internal class loading operations and, for every class loaded after the agent is ready, the JVM calls the agent with the bytecode that represents the class. This bytecode is read to recover the list of internal dependencies and modified by adding those dependencies as metadata using the Bytecode Manipulator. The modified bytecode is resent to the JVM for loading into memory. It is interesting to note that the agent has extremely limited information regarding the origins of the class (e.g., the module it comes from), if it is a duplicate class or not, or by which other classes it will be used. This limits what can be done inside the agent.

The Robusta Java Agent is loaded at startup with the JVM and begins functioning before any application classes are loaded but not before all classes are loaded. Indeed, the JVM loads some classes before loading Java agents such that not all classes can be instrumented by Robusta, hence, not all classes have a complete set of recoverable dependency metadata. This does not affect dynamism because the JVM's core classes are not dynamic or modularized anyway, so the lost dependency information is irrelevant.

It should be noted that Java provides other facilities to obtain information regarding classes and their dependencies, such as Java Reflection, but all the runtime solutions we encountered were insufficient for our needs because they only provide a subset of the classes that have been loaded or a subset of the classes' dependencies. For example, Java Reflection only allows for obtaining the list of fields (each field becomes a dependency), hiding any and all dependencies that exist only within the scope of a method. Furthermore, creating dependency graphs or calculating dependencies before run-time makes it difficult to handle multiple versions of the same class because we cannot know beforehand how the JVM is going to resolve dependencies. Indeed, building the Class Dependency Graph requires recovering the exact dependency resolution

solution that the JVM has chosen and which resides internally (there are no API's to recover this information).

Together, both functions of the agent work to satisfy our open-world assumption and to calculate a complete class dependency graph at any given moment in an unambiguous manner.

9.4.2 Robusta Bytecode Manipulator

The Bytecode Manipulator is charged with reading a bytecode representation of a class and then writing a new version of the class that explicitly details each and every dependency that the bytecode contains. We use ASM for reading and writing bytecode because it is a fast and light library to use.

The manipulation process is a way of creating a special header in each class that contains a list of dependencies in a way that can then be easily accessed at runtime by the Robusta Analyzer. The key to the process is adding the dependency information in such a way that we do not introduce and incompatibilities that can break our application but that still lets us create an unambiguous dependency graph at runtime. To achieve this, we have chosen to augment the bytecode by adding a special @Robusta annotation that contains an array of @ClassDependency annotations. Annotations are by far the superior solution to our problem for multiple reasons:

- they do not modify the behavior of the class,
- they are ignored by other parts of the application that are unaware of the annotation,
- they can hold (some) complex objects like Class objects and arrays.

Furthermore, this approach allows modifying legacy code that has been compiled for earlier versions of Java, before the existence of annotations, as long as the execution platform is a recent version of Java (Java 1.5 or better). This is thanks to Java's retro-compatibility, meaning that bytecode compiled for older versions of Java can run on newer virtual machines (albeit the inverse is not true).

```
package fr.adele.robusta.annotations;

import java.lang.annotation.ElementType;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Robusta {

    public ClassDependency[] value();

}
```

Figure 70: The Robusta annotation.

The `@robusta` annotation that is added to each class can be seen in Figure 70. It simply declares itself as a runtime annotation that holds an array of `@ClassDependency` annotations. The `@ClassDependency` annotation can be seen in Figure 71. For each dependency that is read from the bytecode, one `@ClassDependency` annotation will be added to the `@Robusta` annotation. The interesting part of this annotation is that, for each class that is found, we add that same exact class

to the `@ClassDependency` annotation it not its name or other indirect information. Because this is done at loadtime, the class's name is insufficient to know towards which "real" class the JVM is going to resolve the dependency should there be multiple classes with the same name. By adding the Class object to the annotation, we can query this annotation at runtime and retrieve the exact class to which the dependency holds, unambiguously. Furthermore, once we have the Class we can proceed to calculate the module that loaded it. If there are multiple versions of a class, the dependency only points to one of those versions, which is the version that is obtained through our query.

```
package fr.adele.robusta.annotations;

import java.lang.annotation.*;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface ClassDependency {

    public String className();

    public String modifier();

    public String origin();

    public Class<?> clazz();

}
```

Figure 71: The ClassDependency annotation used to add dependency metadata.

Once the BytecodeAnalyzer has detected and added metadata for each and every dependency, the modified bytecode is ready to be loaded by the JVM (the old version of the class is discarded at this point).

It should be noted that classes loaded through reflection or other detoured mechanisms like `Class.forName()` are not detected as dependencies and thus not added to the dependency metadata. It is possible to add mechanisms to determine these classes, as performed by projects like Tamiflex[106], but this generally requires a more static environment where these dependencies can be pre-calculated, and is beyond the scope of our work.

9.4.3 Robusta Analyzer

The Robusta Analyzer is where dependency graph calculations, such as, calculating the Service Contract, take place. The Analyzer is composed of two parts: the heuristics and algorithms for calculating dynamism and the command-line interface (CLI) used for interacting with the system and for obtaining information regarding the application.

When invoked, the Robusta Analyzer asks the Robusta Agent for the list of classes loaded by the JVM. Once all the classes are available, the analyzer goes through, class-by-class, and reads the @Robusta annotations to recover the dependency metadata. With the classes and their respective dependency metadata, Robusta is able to construct the Class Dependency Graph that is used for

[106] https://code.google.com/p/tamiflex/wiki/Overview

further calculations regarding coupling. To simplify calculations regarding impact analysis caused by the removal of classes and modules, the Class Dependency Graph holds four basic relationships (instead of just two): *Extends*, *ExtendedBy*, *Depends* and *DependendedBy*. This simply allows the graph to be traversed in multiple directions.

Robusta Analyzer also calculates module information. As described above, Robusta runs on the OSGi framework, which uses Java classloaders as the isolation mechanism for creating modules. (Modules in OSGi are called bundles.) For every class that has been loaded, the analyzer calculates the module (i.e., bundle) that loaded it. To do this, we recover the classloader that loaded the class by using the standard Java API and detect if this classloader represents a module or not. If so, then we determine which module it is, and if not, this means that it could be a classloader that is contained in a module or it could be an external classloader that is not contained by any application modules. To find out, we climb the classloader tree until we reach either a module classloader (the classloader tree doesn't usually surpass 3 levels) or until we find the root classloader. After this process, we have all the information regarding dependencies, classes and modules we require for all coupling and Service Contract analyses. Such a computation is only possible because we are running on OSGi which has strict specifications towards how classes are loaded among modules.

Calculating the Service Contract is very straightforward from here. The contract is the transitive set of classes that are reachable following the Extends and Depends relationships. The contract extensions are the set of classes where one class *extends* another inside the Service Contract (see section 6.1.2 for more information). However, OSGi uses Java interfaces to represent services. iPOJO[107] and APAM continue this practice, as does Robusta. This implies that the interface is the starting point for calculating Service Contracts, but not all interfaces represent contracts because not all interfaces are services. Indeed, at the moment we query the OSGi registry for the list of service and recover the service interfaces, but this process is not fully automated and, in our opinion, does not show the interest of the approach. More often, and more intuitively, the user provides the analyzer with the name of the component of the interface to analyze, from where the calculations can commence, hence the interactive nature of the tool. This is slightly more restrictive than a fully automated approach, but developers and architects already know the interfaces that are proposed by the services they have designed and are trying to decouple.

9.4.4 Robusta architectural overview

The Robusta architecture is presented in Figure 72. Applications are packaged into jar files and loaded, class-by-class[108], by the JVM. For each class-load, the JVM calls the Robusta agent which modifies the class to add the dependency metadata and then returns the class to the JVM to be subsequently loaded into memory and used by the application. These annotations preserve the classes' semantics and avoid introducing inconsistencies that can disrupt the application. Robusta reads and modifies all of the application's classes, including the OSGi framework's classes and even most Java runtime classes. We should note that using regular expressions, Robusta allows the filtering of classes that should not be instrumented or read by the bytecode manipulator. This can

[107] Even if iPOJO supports non-interface services, it is widely recommended to always use interfaces.

[108] The Java specification establishes that classloading is done lazily. Only classes that will be required to execute the application are loaded.

be used as an optimization, where we can focus on dynamic classes only, or because the application requires the bytecode to be unaltered (this can be necessary for certain cases like RMI or bytecode signing and verification processes).

Robusta Analyzer is itself a modular application that runs on OSGi. Robusta interfaces with Shelbie, an OSGi Shell, to provide a command-line interface (CLI) for end-users. Interestingly, the analyzer is also modified by the agent and can be inspected to see coupling and other potential dynamism issues (such as duplicated classes) at runtime. When the analyzer is invoked, it uses the agent hook that is provided by Robusta Agent to retrieve the list of loaded classes and dependency information, and then proceeds to reply to the user's request.

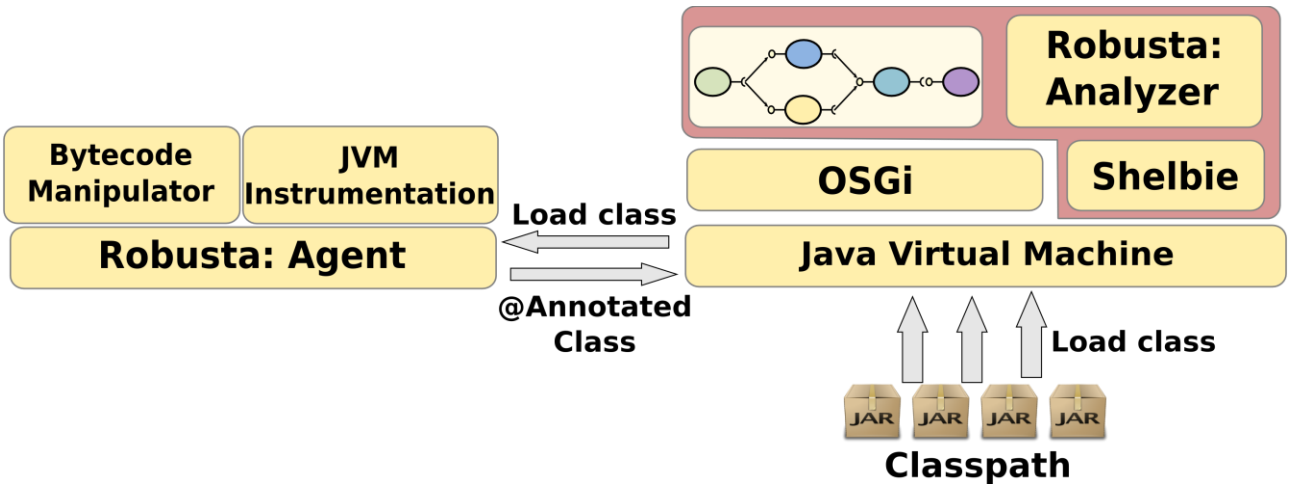

Figure 72: The Robusta high-level architecture

9.4.5 Robusta interactive commands

The OW2 Shelbie project provides end-users with a command line interface and a series of basic commands to interact with the OSGi environment. Robusta implements Shelbie based commands to facilitate the analysis of coupling and the impact of dynamic reconfigurations. Given the runtime nature of our tool, once the application has started, we can interactively invoke coupling analysis commands on running applications. All Robusta commands are namespaced with the `robusta:` namespace. Among the commands available are `robusta:class`, `robusta:duplicates`, `robusta:classloader`, `robusta:graph` and `robusta:service-contract`. The commands are described as follows:

robusta:class [options]	The `class` command provides information for all loaded classes.
Available Options	**Description**
none	Prints class statistics. Same as --stats.
-c, --classes	Prints all classes loaded by the JVM.

-a, --annotated-classes	Dump all classes that have Robusta annotations.
-t [typename], **--annotated-class** [typename]	Dump all annotations for the given type (interface or class).
-T [typename], **--annotated-robusta-class** [typename]	Dump all annotations for the given type (interface or class).
-A, --annotations	Dump all annotations for all classes.
-cl, --include-classloaders	Includes the classes' classloader when printing class information.
-m, --module, -b, --bundle	Include the classes' module/bundle when printing class information.
-gc, --garbage-collection	Instruct the JVM to attempt garbage collection *before* calculating (this does not guarantee GC will be performed but often succeeds).
-h [typename], **--hierarchy** [typename]	Dump the type hierarchy for given type (interface or class).
-all, --all	Dump all information regarding classes.
-sort, --sort	Sort class and interface names alphabetically.
-n, --show-numbers	Show line numbers.
-v, --verbose	Verbose output.
-debug, --debug	Includes debugging output.
-s, --stats	Print stats regarding number of classes and classloaders.
--help	Prints this table of commands.

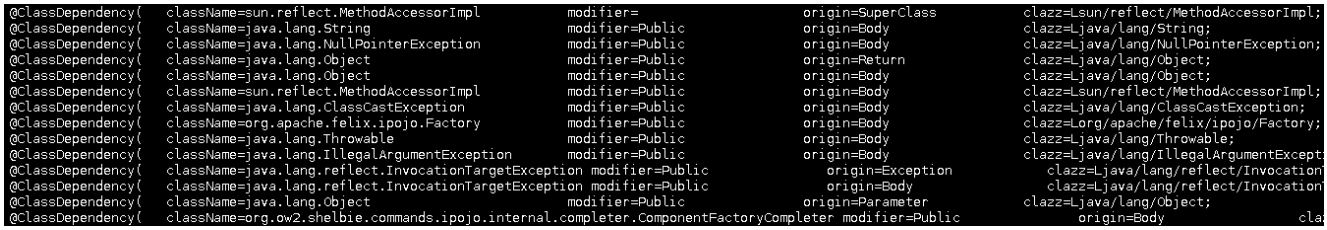

Figure 73: An example of Robusta showing class dependency annotations for the org.ow2.shelbie.commands.ipojo.internal.completer.ComponentFactoryComplete class

robusta:duplicates [options]

The `duplicates` command provides information for all classes and interfaces that are duplicates. A duplicate class is a class that has been loaded by one or more classloaders. The canonical name of the class is used to compare the classes and determine if they are duplicates (e.g., fr.imag.robusta.Test).

Available Options	Description
none	Prints duplicates statistics. Same as --stats.
-d, --duplicates	Print duplicated classes and interfaces using canonical class name to sort.
-D, --duplicates-by-cl	Print duplicated classes and interfaces using classloader to sort.
-t [typename], **--type** [typename]	Print duplicated classes and interfaces that match the given name.
-cl, --include-classloaders	Includes the classes' classloader when printing class information.
-m, --module, -b, --bundle	Include the classes' module/bundle when printing class information.
-gc, --garbage-collection	Instruct the JVM to attempt garbage collection *before* calculating duplicates (this does not guarantee GC will be performed but often succeeds).
-all, --all	Dump all information regarding duplicates.

-n, --show-numbers	Show line numbers.
-v, --verbose	Verbose output.
-debug, --debug	Includes debugging output.
-s, --stats	Print stats regarding number of duplicates.
--help	Prints this table of commands.

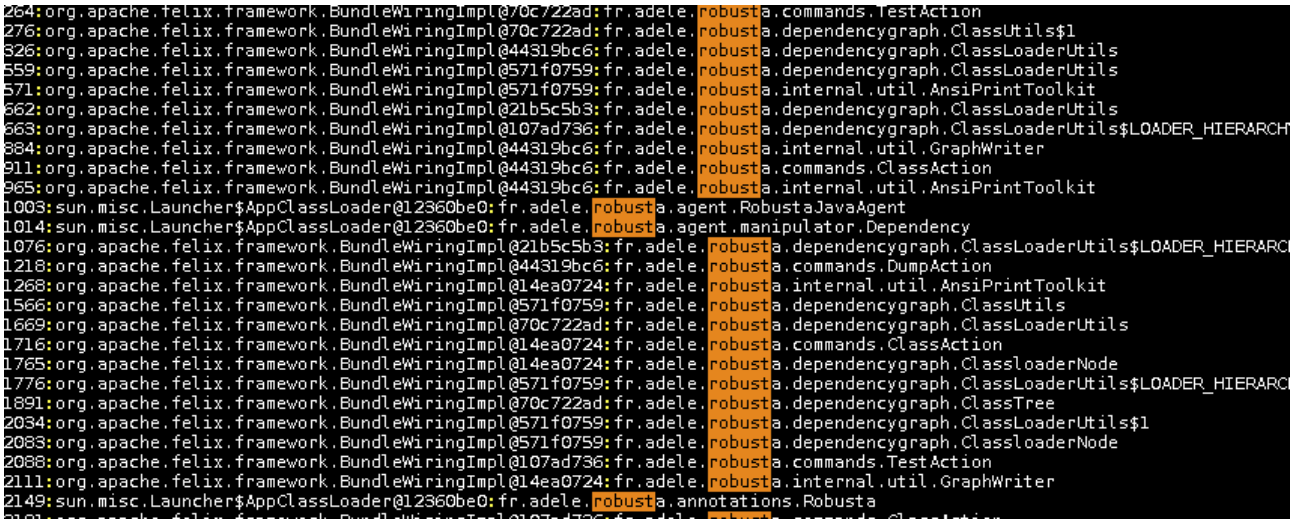

Figure 74: An example of Robusta showing duplicated classes.

robusta:classloader [options]

The `classloader` command provides information for all classloaders. A classloader is used to read classes and provide them to the JVM to be loaded into memory and later instantiated. Classloaders are a good representative for modules because modules use classloader-based isolation (a modules has one classloader).

Available Options	**Description**
none	Prints classloader statistics. Same as --stats.
-t, --classloader-loading-tree	Print classloader tree (using how they were loaded by one another).
-T,	Print classloader tree (using how they delegate to one

--classloader-delegation-tree	another).
-l, --list, --classloader-list	Print classloader list (table).
-m, --module, -b, --bundle	Include the classloader's module/bundle information when printing.
-gc, --garbage-collection	Instruct the JVM to attempt garbage collection *before* operations (this does not guarantee GC will be performed but often succeeds).
-all, --all	Dump all information regarding classloaders.
-n, --show-numbers	Show line numbers.
-v, --verbose	Verbose output.
-debug, --debug	Include debugging output.
-s, --stats	Print stats regarding number classloaders.
--help	Prints this table of commands.

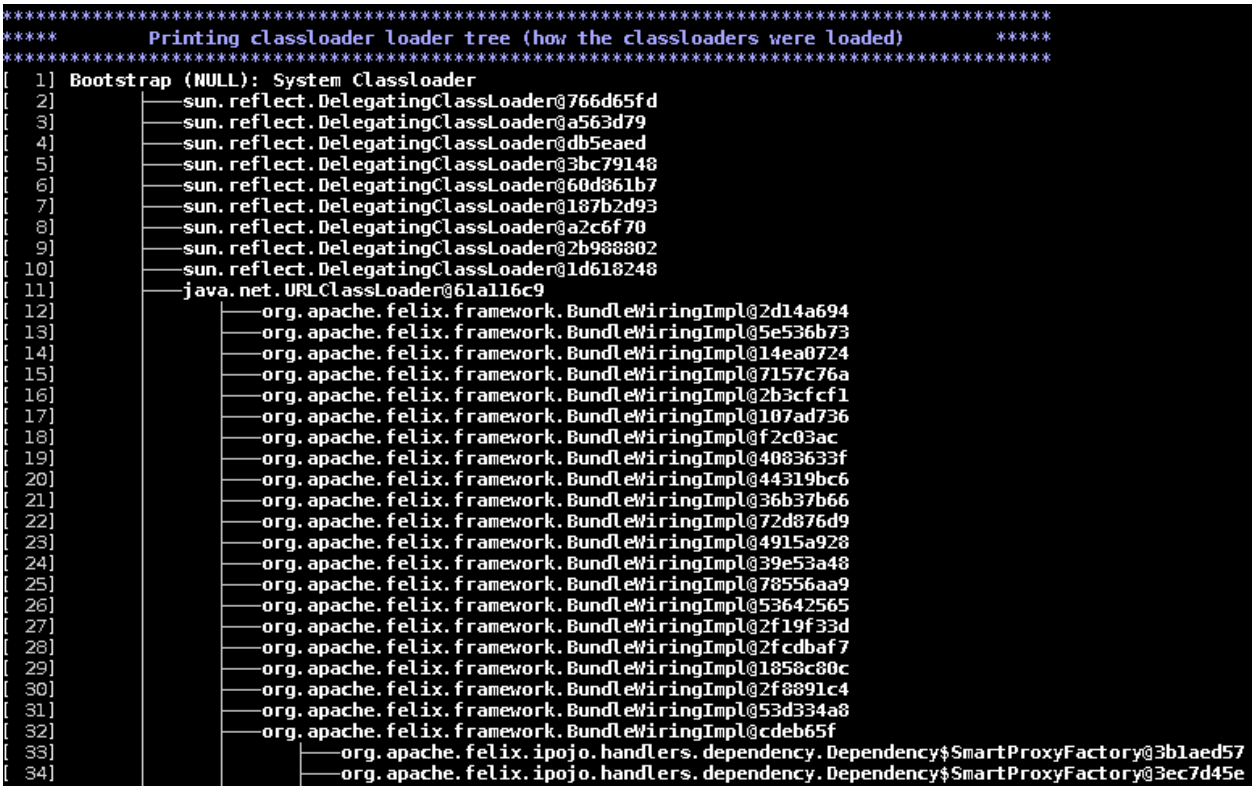

Figure 75: An example of Robusta showing a classloader tree.

robusta:service-contract [options] `service-interface`

The `service-contract` command calculates and outputs the service contract and contract extensions for the given service interface. If provided a class instead of an interface, the same transitive dependency graph calculations are performed.

Available Options	Description
None	Same as –service-contract.
-e, --extensions	Outputs the service contract's extensions only. If multiple extensions are available it prints them one-by-one.
-sc, --service-contract	Outputs the service contract but not the extensions. This is the default operation.
-esc, --extended-service-contract	Outputs the service contract and then the service contract extensions.

-m, --module, -b, --bundle	Changes the output to print modules instead of classes.
-ct, --component-type	When this option is given the service-interface is interpreted to be a component type instead. The service-contract is then calculated once for each service that the component-type provides and requires.
-debug, --debug	Include debugging output (does not change file).
--help	Prints this table of commands.

robusta:graph [options] `filename`

The `graph` command writes the desired graph to a file specified by filename that can be used externally. If filename is missing the graph is printed to console.

Available Options	**Description**
none	Same as --dot --classes.
-d, --dot	Writes the desired graph to filename in the DOT graph description language. The graph written is a directed graph (digraph). The is the default.
-g, --graphml,	Writes the desired graph in the graphml graph description language.
-c, --classes	Print a directed graph of class dependencies.
-m, --module, -b, --bundle	Print a directed graph of module dependencies.
-cl, --classloader-loading-tree	Print a directed graph of classloaders using how they were loaded by one another.
-CL, --classloader-delegation-tree	Print a directed graph of classloaders using how they delegate to one another.
-cm, --classes-and-modules	Print a directed graph of module dependencies with subgraphs representing class dependencies. (This type of graph does not scale well.)

-f, --force	Force file overwrite
-v, --verbose	Verbose output (does not change file).
-debug, --debug	Include debugging output (does not change file).
--help	Prints this table of commands.

robusta:stats [options]

The `stats` command prints statistics regarding classes, classloaders, modified classes by robusta (annotated classes), modules and duplicate classes. Equivalent to running robusta:command –stats except more information is put into a single table.

Available Options	**Description**
none	Prints statistics.
-v, --verbose	Verbose output (does not change file).
-debug, --debug	Include debugging output (does not change file).
--help	Prints this table of commands.

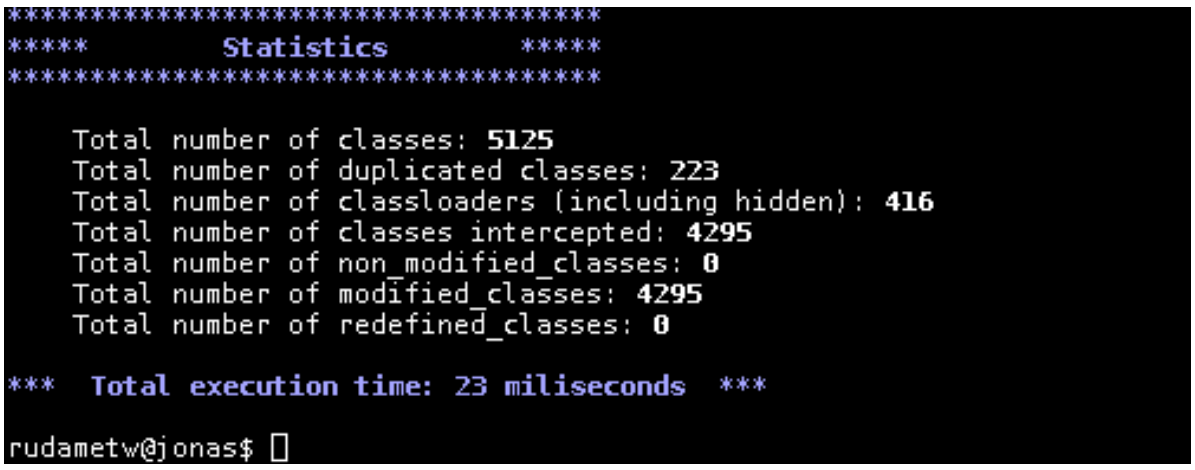

Figure 76: An example of Robusta showing statistics on classes and classloaders.

9.5 Experimentation

We have implemented and successfully tested Robusta using different target applications. Given our use of underlying frameworks, our initial tests with Robusta were, interestingly, done using Robusta itself and all of its dependencies, such as, Apache Felix OSGi, Apache Felix iPOJO, OW2 Shelbie and OW2 Chameleon. The framework to run Robusta currently consists of 33 modules and provides over 3 thousand classes.

Nevertheless, we tested Robusta mainly using two different projects. The first project is a small TODO List application that we developed and which uses a very small number of classes and modules. This allowed us to test Robusta for usefulness and correctness in a controlled fashion with a codebase we could manually inspect and modify. Our other main test case was OW2 JOnAS, which is used for industrial applications and provides a much larger test case that is particularly useful for calculating overhead and to detect any inherit problems with Robusta.

9.5.1 TODO List using ROSE

ROSE[109] is part of the OW2 Chameleon project and aims to smooth the design and execution of an application composed of distributed services. ROSE integrates various protocols and frameworks into a uniquely dynamic and extensible framework for building distributed systems. ROSE has been designed from the ground up to operate with OSGi and iPOJO, benefitting from their levels of modularity and dynamism, allowing ROSE to dynamically add and remove different distributed communication protocols at runtime, such as, web services, REST, JSON-RPC and XML-RPC.

We have implemented a small TODO list application using ROSE. Our goal was to test Robusta using this application. In Figure 77 we present our class diagram for the implementation of `Todo`s and our `TODOList` interface for our `TODOList` service.

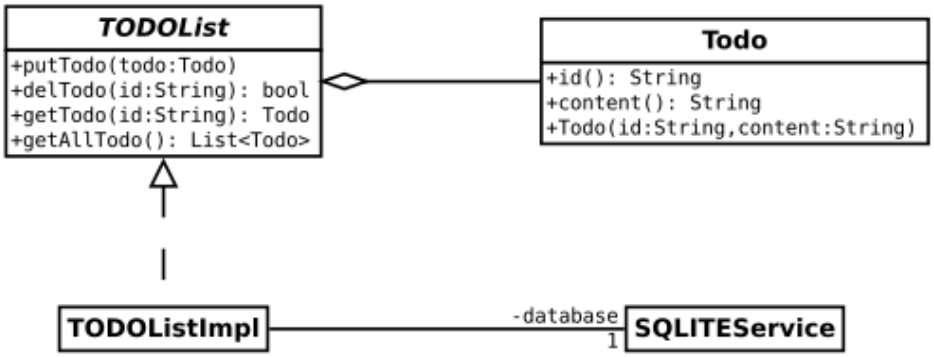

Figure 77: Class diagram of the TODO list example application.

[109] http://wiki.chameleon.ow2.org/xwiki/bin/view/Main/Rose

ROSE is used to export the TODOList service as a REST[110] endpoint and uses the JSON service for serialization and deserialization. Clients can access the TODOList service through simple http requests[111]. Three types of requests are supported: PUT, GET and DELETE HTTP requests, which respectively add, retrieve and delete TODOs from the TODOList. Table 3 illustrates the REST API that is created.

URL	Request type	Description
`/todolist`	GET	Returns a JSON array of TODOs.
`/todolist/{id}`	GET	Returns the TODO item that matches id.
`/todolist`	PUT	Adds the content of the request into the TODO list.
`/todolist/{id}`	DELETE	Deletes the TODO item that matches id.

Table 3: The TODO list application's REST API.

The architecture for our application is shown in Figure 78 and consists of 3 components, two provided services, one remote client and a remote HTTP Rest service. The `RESTTodo` component requires a `TODOList` service and implements the necessary functionality for ROSE and the REST metadata to export the service. The `TODOListImpl` component provides the `TODOList` service and the basic TODO list functionality shown in Table 3. The `TODOListImpl` component requires a backend to store TODOs. The specific service required is the `DataSourceFactory` service, which is provided by the SQLite modules that have been packaged in OW2 Chameleon.

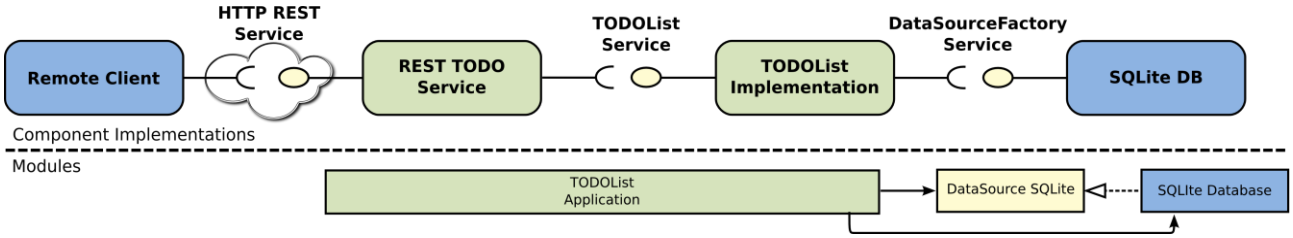

Figure 78: Initial TODO List components and modules.

Our initial packaging placed all of our TODO application into a single module, including the definition of both TODO List components. We proceeded to modularize our application around our components using Robusta. The objective was to allow components, the `REST TODO Service` or the `TODOList implementation`, to change dynamically, without impacting other components at runtime.

To achieve decoupling, we proceeded to analyze the `TODOList Service` interface and place it into a separate module that would represent the service contract. Once we found the classes, in this case two classes, we proceeded to separate the component implementations into their own

[110] The application does not aim to implement the full REST approach. So HATEOS and other characteristics were voluntary ignored.

[111] For simplicity, TODO items cannot be updated.

packages as well. Because we did not extend classes in our application, there was no need to create contract extensions modules. Robusta properly calculated the dependency graphs each time and facilitated the selection of new classes for packaging. Robusta also found a class dependency that extended from our application to the `SQLite Database` module. Figure 79 shows the result of our changes to the TODO List application. We decomposed our application into 3 modules, two of which can now evolve independently (the Service Contract cannot change without impacting the other modules).

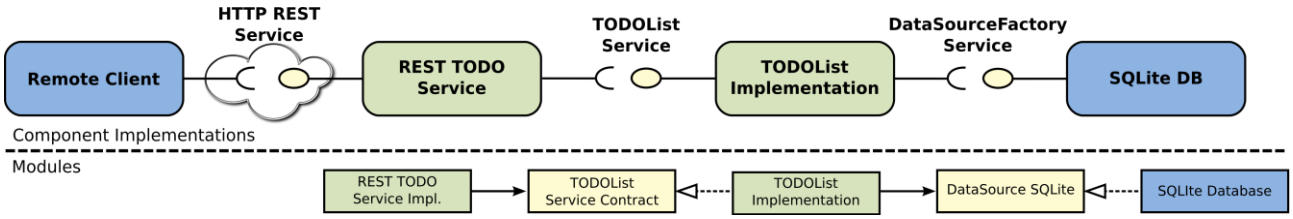

Figure 79: Resulting TODO List architecture after decoupling analysis and packaging changes.

To test our application and its newly acquired dynamism, we proceeded to update, stop, start and replace our modules at runtime. Robusta verified that no duplicated classes resulted from these operations, meaning that they were properly decoupled. Robusta proved its usefulness and its ability to help architects and developers to modularize their applications.

9.5.2 OW2 JOnAS Java Enterprise Edition Application Server

JOnAS is a Java EE 5 certified open source application server built and hosted by OW2. As a project, JOnAS started in 1998. The current version of JOnAS is built on top of OSGi, and as such, provides and exploits the same levels of dynamism as do other OSGi projects, and allows for the integration of OSGi applications and component models such as iPOJO. JOnAS provides clustering and high availability mechanisms, Web Services, Java EE Connectors, LDAP access, IIOP and many other features.

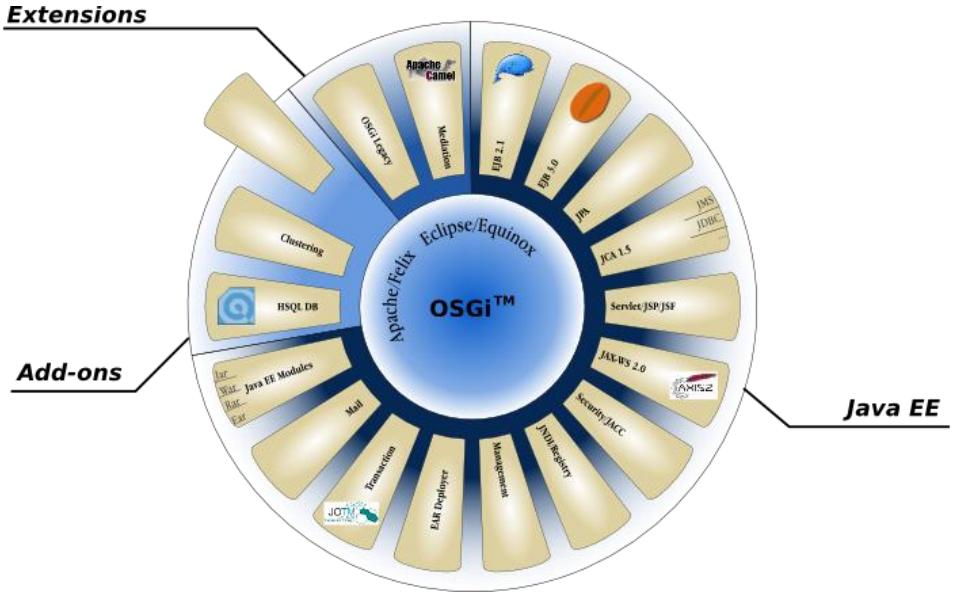

Figure 80: JOnAS's services overview. JOnAS provides a-la-carte services that are dynamically deployed.

According to Ohloh[112], JOnAS is a high activity open source project with multiple contributors and consists of 9.87 million lines of code. According to Ohloh's effort analysis using the COCOMO model, JOnAS is a project that accounts for 3,023 years of effort. However, we believe this to be exaggerated given the way Ohloh counts lines of code and the way the OW2 consortium contributes across project boundaries. The JOnAS team reuses and contributes to many OW2 projects (e.g., EasyBeans, JORAM, Shelbie, OW2 Utils) and integrates them into a single application server, making it difficult to estimate its size. Our estimates for active lines of code managed by JOnAS range from +400 thousand lines of code to 1 million, depending on which OW2 projects are included. A final release version of JOnAS undoubtedly holds much more code that is provided by other open source projects and communities. We estimate this to be around 3 million lines of code in a single release.

Criteria tested	**Result**
Startup time (w/o robusta)	43 seconds
Startup time (with Robusta)	52 seconds
Runs properly with Robusta	Yes. Does not show problems caused by class interception and transformations. However, JOnAS was not extensively tested. Startup, command line access, and administration console were tested.
Number of classes intercepted and transformed	8324 classes in total.
Time to calculate Class Dependency Graph	20 - 163 milliseconds.
Time to manipulate a class	0 – 220 milliseconds. Slower speeds at startup because the system is loaded. Average is ~9ms.
Time to calculate Service Contract	3 - 20 milliseconds in addition to the time needed to calculate the Class Dependency Graph time.
Time to calculate Extended Service Contract	3 - 64 milliseconds in addition to the time needed to calculate the Class Dependency Graph time.

Table 4: Results obtained when testing Robusta with OW2 JOnAS.

JOnAS is by far the largest application we tested Robusta with and also the most challenging. We tested Robusta on version 5.3.0-RC1, which consisted of over 300 modules and 120 iPOJO components. Given the sheer size of JOnAS and the effort required to decouple components in JOnAS, our tests were limited to introspection and checking Robusta's scalability and overhead.

[112] http://www.ohloh.net/p/jonas

Our results conclude that Robusta does not inhibit the application server from functioning properly and has little overall overhead. Furthermore Robusta's interactive nature ensures there is very little overhead if Robusta is not in use. These data can be seen in Table 4.

In order to execute JOnAS with Robusta enabled, we modified the `jonas` script that is used to start and stop the server (among other functions), and added the Robusta Agent to the command line arguments. For Robusta Analyzer to function, we manually added the required modules to the internal OSGi framework.

This experiment has proven that the collection of the data and the analysis is fast enough even on large software. The graph computation times are somewhat meaningless in interactive mode because the user is effectively much slower than Robusta. However, for automated analyses, these times become more important and should still be quite acceptable.

9.5.3 Graphical output of Class Dependency Graphs

During the design and development of Robusta, we found it desirable to provide a graphical means of viewing an application's complexity. Complexity often remains an abstract concept and is difficult to grasp. Nevertheless, graphical tools for class dependency analysis are few and far between. Yet, because Robusta has access to such information, we proceeded to create directed dependency graphs and visualize them using graph software. The simple fact of obtaining this information and exporting it to other tools can provide insight into an application

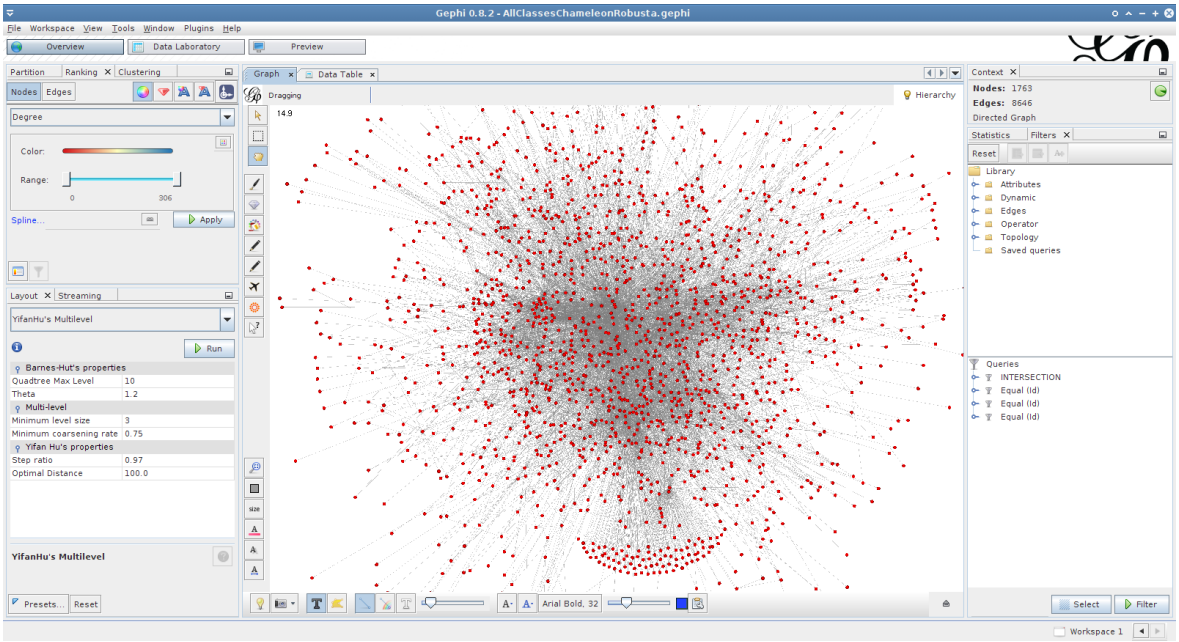

Figure 81: Shows a screenshot of Gephi with a 3000+ class dependency graph.

Robusta can export dependency graphs to files using the `robusta:graph` command. Such graphs can be visualized using various software, of which we tested three: yEd[113], Graphviz[114] and

[113] http://www.yworks.com/en/products_yed_about.html
[114] http://www.graphviz.org/

Gephi[115]. Each tool has its advantages and defects, but we largely preferred Gephi. Graphviz provides multiple algorithms and outputs image files (e.g., png) after processing, but few of the algorithms can handle several thousand nodes, and the ones that can appear to be unusable. yEd is quick and light but showed rendering issues. Gephi was the only tool that allowed consistent viewing and manipulation of multi-thousand node class dependency graphs.

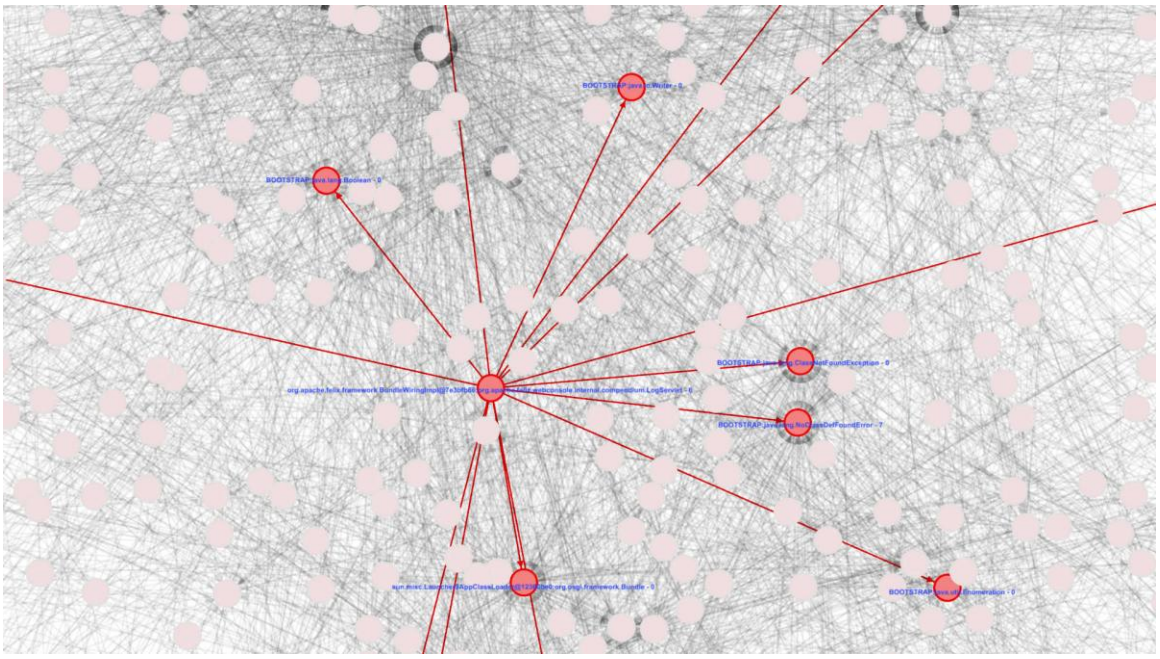

Figure 82: Shows a close-up of a class and its dependencies in Gephi.

Disappointingly, none of the graph visualization software provided adequate support for clusters. This means that it was not possible to wrap groups of classes into larger nodes that represented the modules they were contained in. To circumvent this issue, Robusta exports multiple independent graphs for classes, modules and classloaders.

9.5.4 Results & Lessons

The execution overhead of using Robusta is low, as can be seen in the tests using JOnAS. The average graph calculation times are under 100 milliseconds, and the time to instrument a class with the Robusta annotations is often under 10 milliseconds. Furthermore, there is no detectable execution overhead when Robusta is not used, i.e., when there are no dynamic events in the application.

However, memory overhead is much more difficult to calculate because of the complexity of the inner workings of the Java VM and of the operating system. We have pagination, shared libraries, caches, and other features that often make such calculations nonsensical. As such, we have not attempted to estimate this overhead in practice. Be that as it may, the Robusta Agent does cause a permanent memory overhead for each class and each dependency that is encountered. Table 5 shows Robusta's memory overhead.

[115] https://gephi.org/

Permanent Memory Overhead cause by Robusta Agent
1 annotation for each class (`@Robusta`)
1 annotation per class-dependency (`@ClassDependency`)
4 attributes per class-dependency annotation (`String`, `String`, `String`, `Class`)

Table 5: Robusta memory overhead.

Among the lessons we have learned are that Java class loading is lazy and this can impact dynamism and coupling calculations. Loading classes late in the execution can still cause hidden coupling to occur, which can penalize dynamism and result in undesirable behavior. To properly test an application you have to execute it thoroughly in order to cause all classes to be loaded. Furthermore, garbage collection is also lazy, meaning that there is no guarantee that classes will be timely collected and freed from memory. This makes debugging difficult because, for Robusta, duplicate classes are often a good indicator that something is not working properly. If garbage collection has not occurred, then every update results in duplicated classes.

Interestingly, duplicate classes do occur even if there are no issues with dynamism or with the application. This is because large software tends to repackage libraries of different versions. Refactoring code and using a single version of a library is often too costly or tedious and provides little immediate benefit to the application. This can lead to issues if not properly managed. Luckily, Robusta helps detect these issues.

Using the root hierarchy object (the Object class in Java) can be very problematic for dependency analysis because it opens the service to being contaminated by any object in the virtual machine. If we are pessimistic in our calculations, we should suppose that such a service is coupled to everything on the platform. If we are optimistic and suppose that it is decoupled, this may lead to memory leaks or undesirable behavior (e.g., class cast exceptions). The use of the root hierarchy object should be avoided.

Decoupling can be costly to developers and to maintenance aspects of software because it adds to the number of modules required. It tends to be expensive because current tools do not make it easy or automate the process sufficiently. Separating provider implementations from interface modules, and consumer implementations, and service extensions, all add more and more modules that need to be maintained. The current state of software development does not sufficiently support the developer when making fine-grained dynamic software.

Finally, there is a tendency to orient coupling from consumer components to provider components. Many projects place the Service Interface inside the provider's modules in order to reduce the number of modules. This makes the consumer's dynamic but changing a provider can have an extensive impact on the running system. We feel that developers should move to implementing fully independent Service Contracts that are maintained separately from both provider and consumer components.

9.6 Conclusion

We have implemented and tested Robusta to prove the feasibility of our approach using current development techniques. Robusta can be used as an interactive tool to detect coupling and to improve dynamism in modular applications. We have put Robusta's capacity to detect coupling to the test in complex applications that are used in industrial settings. We have also shown that the overhead caused by Robusta is minimal and acceptable in most situations. The interactive nature of the current iteration of this tool means that it can be used primarily for diagnostics and debugging. Robusta creates dependency graphs that represent the state of the application unambiguously.

The Robusta prototype is a proof-of-concept that the approach proposed in this thesis can be used even when applications execute in an environment that follows Open-World assumptions. The complexity involved in decoupling components in dynamic applications justifies our tool because there is a real need for achieving safe-dynamism through decoupling. Robusta is useful for improving the understanding of dynamic applications. We have shown that complexity is an issue when handling dynamism and that dynamism can be tedious. After having demonstrated Robusta to software architects and OSGi experts, we feel that Robusta is, in its current form, a step forward for the analysis of dynamic applications but there is still much work to do.

Part IV:

Conclusions

Chapter 10

Conclusions & Perspectives

"Data is not information, information is not knowledge, knowledge is not understanding, understanding is not wisdom."
—Clifford Stoll

"A human being should be able to change a diaper, plan an invasion, butcher a hog, conn a ship, design a building, write a sonnet, balance accounts, build a wall, set a bone, comfort the dying, take orders, give orders, cooperate, act alone, solve equations, analyze a new problem, pitch manure, program a computer, cook a tasty meal, fight efficiently, die gallantly. Specialization is for insects."
—Robert A. Heinlein, "Time Enough for Love"

10.1 Summary & Discussion

Developing dynamic applications is hard, complex, and error-prone, and there is a general lack of guarantees, guidelines, best practices and tools to assist us in doing so. Despite that, dynamism is becoming a growing concern as more and more applications and domains are pushing for it. Anecdotally, the current software market is very competitive and mastering dynamism would provide a definite edge. Dynamism is the next step taken after modularity, which allows us to tame complexity. Dynamism is—very informally—modules on steroids. However, we have shown that current approaches are insufficient for building robust dynamic applications that remain consistent despite dynamism. This is indeed problematic and, arguably, one of the main reasons for the lack of adoption of dynamic component models in real world applications. It is our view that there are few types of applications that would sacrifice consistency to achieve dynamism, and there is still a general lack of awareness to all the concerns involved in building dynamic applications and the extent to which dynamism is invasive and cross-cutting.

Our approach is a strong step towards ensuring consistency in dynamic applications. We propose guidelines to properly decouple component implementations and instances in order for them to remain consistent, avoid corruption, and recover from potential corruption in dynamic environments. We allow components to use complex behaviors of interaction which, in most approaches that ensure consistency, are not allowed. We make it possible to create highly resilient dynamic components. In addition, we have elevated dynamism to the architectural level, where it can be analyzed, better understood and reasoned about in a systematic way. We have found that dynamism can also be highly selective and targeted, allowing applications to implement

dynamism and dynamic-decoupling where the application can benefit from them the most. At the architecture level, dynamism decisions can be put to the test, corruption can be calculated, and hypotheses and assumptions made by architects can be verified. Architects can build applications to resist and handle expected dynamism.

We have created various concepts that assist architects in taming and managing dynamism. We use a highly generic component zone concept that allows architects to control the degree to which dynamism is enforced or allowed to corrupt and contaminate parts of the architecture. In order for dynamism to be harnessed, we require more information about the application and about the assumptions the components have been developed with. Yet, it is already an important best practice for this information to be well documented. Making this information available for design-time analysis and runtime management is a relatively small effort, no different than the effort to properly document source code.

At runtime, the application is meticulously controlled. We allow for proactive and reactive changes, permitting a large array of dynamic applications to be built with our approach. Furthermore, and orthogonally, our dynamic applications can tolerate both expected and unexpected dynamism, while ensuring consistency in both cases. To our knowledge, the more complicated case to handle is unexpected reactive dynamism—caused by devices, remote services, networked connections, random failures—which we manage and still ensure the application remains consistent (it is possible we invalidate every component to ensure consistency though). Additionally, the runtime strives to minimize the impact of dynamic change on active requests. Our safe-stopping algorithm exploits passivation, branching and corruption analysis in an attempt to never stop the application.

An important part of our reflection has been placed on tooling. The need for tools for assistance and verification of dynamic application is real. We need to guarantee that our applications are not silently corrupted because of otherwise inoffensive dynamic changes. This requires, as we have shown, a gradual shift in procedure, making dynamism a central concern in the development of modern applications. Of course, adding the "dynamism concern" on top of many other concerns not addressed in this work increases the excessive amount of complexity that developers are already required to handle. We feel that simply identifying the problems caused by dynamism and adding dynamism to applications is not enough. This virtually guarantees that mistakes and errors will be made, and applications will behave undesirably. We believe that tooling is a potential solution for mitigating the complexity in building large dynamic applications. Tooling has been shown to increase productivity and understanding, such that, we believe dynamism can be integrated into current development methodologies with the right tools for developing, testing and managing it.

10.2 Perspectives

Our work proposes an approach to building dynamic applications. Nevertheless, there is still much to do to improve and integrate dynamic applications. In this section we explore some of the perspectives that we have uncovered.

10.2.1 Integrated Design Environments

Dynamism and component-based software development would be well served by an integrated design environment that uses architectural concepts that can be directly manipulated by developers and architects. Indeed, domain specific development environments are one possible solution to providing an integrated environment capable of designing, developing, debugging and monitoring dynamic applications. In the particular case of dynamism, this would allow tools to analyze the impact of dynamism to be directly involved in the development process, making dynamism concerns visible at multiple levels, allowing developers and architects to both quickly understand and modify the application to better achieve their goals.

There is still a large amount of work to do in regards to assisting and verifying dynamism in component-based applications. As we have seen in this thesis, static code analysis and dynamic runtime analysis are both useful in characterizing components and their dynamic behavior to help developers. A testing environment that provides the necessary feedback to developers and architects, integrated into the development environment, would allow for the prompt integration of evolving dynamism concerns.

10.2.2 Dynamism in languages, compilers and virtual machines

A logical step towards handling dynamism is to integrate it into all levels of development and execution. Indeed, making dynamism a central concept, not only at the architectural level, but also in the programming languages, compilers and virtual machines would make it more accessible, central to development and better handled. Furthermore, there are potentially many improvements that could be made in regards to instrumentation, coupling detection, performance and monitoring. Interestingly, many of these operations and calculations are already performed internally by virtual machines but they are inaccessible to application developers. Providing APIs that can access this information and integrate it into high-level decisions made by architects seems promising. There are some ad-hoc instrumentation mechanisms, such as JMVTI[116], that we have yet to fully explore but might provide some of the functionality we are looking for.

Of course, there is still a lack of basic concepts such as component, module, service and many others, that need to be handled too. Ideally, the integration of such concepts into the language would vastly simplify the development of dynamic applications. In the meantime, we believe compilers could improve verifications and static analysis performed on component-based dynamic applications. Indeed, there is a strong separation between virtual machines and component frameworks in current approaches that is only partially mitigated by the use of generics, annotations and external metadata. Integrating component frameworks directly into virtual machines would improve the current situation.

Nevertheless, given our current concepts, we believe that there is a lot of low-hanging fruit in this area that is ready to be exploited and wouldn't require diving too deep into these underlying technologies. Homogenous end-to-end concepts for component instance and implementation, service, service contract, service interface, free and managed objects, and modules that are

[116] http://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/

understood and verified by the underlying virtual machines and compilers would do a great service to promoting dynamism and to ensuring necessary guarantees such as consistency.

10.2.3 Fuzzy error detection and Failure Oblivious systems

In sections 5.4 and 7.1 we have described the mechanism that we use to decide which failures merit architectural reconfigurations and which do not. Furthermore, we have already mentioned some of the limitations of failure detection in our approach and how a more complete failure detection system would complement our work.

There is interesting work in the area of failure oblivious systems [Rinard *et al.* 2004] that could be integrated into our work. Notably, failure is often seen as binary yet, in many cases, there are many shades of gray in between. Furthermore, some work has shown that, in the case of a fault, simply providing "something" is often good enough to continue execution and avoid shutting down the system. For example, should a component require a temperature that is returned using an integer, and the temperature device fails, we could simply return "any" integer value and continue execution. The notion of *approximate computation* refines this issue by, ideally, providing a value approximate to the one that should have been provided, within a given margin of error. The risk of relaxing the consistency or coherency constraints of the application and "doing something", instead of simply failing, is that the system might perform undesirably because of corruption. If, for example, the replaced temperature value was then used to decide if we turn the oven on or off, we would probably not want to be sending "any" value. However, if the temperature value was provided by one sensor in ten thousand in a building complex, the approximate value would probably be sufficient and preferable to failure.

We could introduce the concepts of relaxed consistency and approximate computing into an enhanced failure detection mechanism, largely increasing the range of configurable behaviors that the system would exhibit. Furthermore, this would play down our strict concept of consistency in favor of increased availability and resilience.

10.2.4 Autonomic computing

As we have seen, our approach is based on mechanisms that are *"strategy free"*. The Robusta framework is itself well isolated from APAM and iPOJO, and relies on fairly generic concepts like component and contract. Furthermore, Robusta is not "intelligent" and does not perform smart or anticipated actions.

In sharp contrast, autonomic computing's goal is to produce autonomous software applications that administer themselves. Autonomic applications are applications that are managed by autonomic managers. Autonomic applications are particularly interesting and potentially useful in the case of context sensitive, ubiquitous, and cloud-based applications. This is because these applications are constantly striving to adapt and optimize themselves at runtime depending on their context. For these types of applications, the underlying execution framework is particularly important and must reify the necessary concepts for the autonomic manager to adapt the application.

Robusta is particularly promising because it provides a solution to the problem of maintaining consistency given dynamic change. Not only that, it supports many types of dynamic change. Autonomic managers can use Robusta—whose job is to transparently passivate and silently remove unused components from the architecture—while specializing on higher level strategies and optimizations that can be performed. Indeed, we believe that Robusta can be considered, to a certain extent, an Autonomic Manager *enabler*.

Among the possible functions autonomic managers can have, we feel that resource accounting, context prevision and adaptation, and quality of service concerns are important, to mention a few. Using Robusta, an autonomic manager can specialize in optimizing the application by adapting its architecture at runtime, while not worrying about the underlying changes. In a Java EE server cluster, the autonomic manager can focus on maximizing CPU and other resource use by deploying new components and features, removing old components, and directing requests to pertinent servers (*e.g.*, load balancing). Robusta ensures that the changes are applied in a timely and consistent manner.

Another interesting use-case for autonomic managers is in detecting failing or failed components. (Byzantine faults are particularly difficult to detect.) The autonomic manager can use many different techniques to detect failing or faulty components, such as response times, response values, memory or CPU use, and then decide to reboot components. Micro-reboots have shown promise in making applications more resilient. Because Robusta allows for many types of dynamic changes, micro-reboots being no different than updates or substitutions, this could be handled transparently by Robusta, while ensuring consistency.

Bibliography

Abdurazik, A., 2007. *Coupling-Based Analysis of Object-Oriented Software*. PhD Thesis, George Mason University, Fairfax, VA, USA.

Aguilera, M.K., Mogul, J.C., Wiener, J.L., Reynolds, P. and Muthitacharoen, A., 2003. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review*, 37(5), p.74.

Aiken, M., Fähndrich, M., Hawblitzel, C., Hunt, G. and Larus, J., 2006. Deconstructing process isolation. In *Proceedings of the 2006 workshop on Memory system performance and correctness - MSPC '06*. New York, New York, USA: ACM Press, p. 1.

Aldrich, J., Chambers, C. and Notkin, D., 2002. *ArchJava: connecting software architecture to implementation*, New York, New York, USA: ACM Press.

Allen, R., Douence, R., Garlan, D. and Cmu, C.M.U.I., 1998. Specifying and Analyzing Dynamic Software Architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98), Lisbon Portugal*. pp. 1–15.

Allen, R. and Garlan, D., 1997. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3), pp.213–249.

Allen, R.J., 1997. *A Formal Approach to Software Architecture*. CarnegieMellon University.

Andrade, L., Fiadeiro, J., Bernardo, M. and Inverardi, Paola, 2003. *Formal Methods for Software Architectures* M. Bernardo & Paola Inverardi, eds., Berlin, Heidelberg: Springer Berlin Heidelberg.

Barais, O. and Duchien, L., 2005. SafArchie Studio: An ArgoUML extension to build Safe Architectures. In P. Dissaux, M. Filali-Amine, P. Michel, & F. Vernadat, eds. *Architecture Description Languages*. New York: Springer-Verlag, pp. 85–100.

Barais, O., Franc, A., Meur, L., Duchien, L. and Lawall, J., 2008. Software Architecture Evolution. In T. Mens & S. Demeyer, eds. *Software Evolution*. Springer, pp. 233–262.

Baresi, L. and Ghezzi, C., 2010. The disappearing boundary between development-time and run-time. In *Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER '10*. New York, New York, USA: ACM Press, p. 17.

Baresi, L., Heckel, R., Thone, S. and Varro, D., 2004. Style-based refinement of dynamic software architectures. In *Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*. IEEE Computer Society, pp. 155–164.

Bass, L., Clements, Paul and Kazman, Rick, 2003. *Software Architecture in Practice*,

Beck, K., 2000. *Extreme Programming Explained: Embrace Change*, Addison Wesley.

Belady, L.A. and Lehman, M., 1976. A model of large program development. *IBM Systems Journal*, 15(3), pp.225–252.

Bennett, K.H. and Rajlich, V.T., 2000. Software maintenance and evolution. In *Proceedings of the conference on The future of Software engineering - ICSE '00*. New York, New York, USA: ACM Press, pp. 73–87.

Bergland, G.D., 1981. A Guided Tour of Program Design Methodologies. *Computer*, 14(10), pp.13–37.

Bidan, C., Issarny, V., Saridakis, T. and Zarras, A., 1998. A dynamic reconfiguration service for CORBA. In *Proceedings. Fourth International Conference on Configurable Distributed Systems (Cat. No.98EX159)*. IEEE Comput. Soc, pp. 35–42.

Bloch, J., 2008. *Effective Java* 2nd Editio., Addison-Wesley.

Bloom, T. and Day, M., 1993. Reconfiguration and module replacement in Argus: theory and practice. *Software Engineering Journal*, 8(2), pp.102–108.

BOYAPATI, C., LISKOV, B. and SHRIRA, L., 2003. Ownership types for object encapsulation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New Orleans, Louisiana: ACM, pp. 213–223.

Bradbury, J.S., Cordy, J.R., Dingel, J. and Wermelinger, M., 2004. A survey of self-management in dynamic software architecture specifications. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*. New York, NY, USA: ACM, pp. 28–33.

Brooks, F.P., 1995. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition)*, Addison-Wesley Professional.

Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V. and Stefani, J.B., 2006. The fractal component model and its support in java. *Software-Practice and Experience*, 36(11), pp.1257–1284.

Buckley, J., Mens, T., Zenger, M., Rashid, A. and Kniesel, G., 2005. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5), pp.309–332.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M., 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, Wiley.

Böszörményi, L., Gutknecht, J. and Pomberger, G., 2000. *The school of Niklaus Wirth: the art of simplicity*, Morgan Kaufmann.

Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G. and Fox, A., 2004. Microreboot -- A Technique for Cheap Recovery. In *Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI)*. San Francisco, CA, p. 14.

Cervantes, H. and Hall, R., 2004. Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model.

Clements, Paul, Garlan, D., Little, R., Nord, R. and Stafford, J., 2002. *Documenting software architectures: views and beyond*, Addison-Wesley Professional.

Cook, S., Harrison, R., Lehman, Meir and Wernick, P., 2006. Evolution in software systems: foundations of the SPE classification scheme. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(1), pp.1–35.

Costa, F.M., Provensi, L.L. and Vaz, F.F., 2006. Using runtime models to unify and structure the handling of meta-information in reflective middleware. In *MoDELS'06 Proceedings of the 2006 international conference on Models in software engineering*. Heidelberg, Germany: Springer-Verlag Berlin, pp. 232–241.

Crnkovic, I., Stafford, J. and Szyperski, C., 2011. Software Components beyond Programming: From Routines to Services. *IEEE Software*, 28(3), pp.22–26.

Cuesta, C.E., De la Fuente, P. and Barrio-Solárzano, M., 2001. Dynamic coordination architecture through the use of reflection. In *Proceedings of the 2001 ACM symposium on Applied computing - SAC '01*. New York, New York, USA: ACM Press, pp. 134–140.

Darwin, C., 1859. Origin of Species. *Library*, 138(1), pp.1854–1854.

David, P.-C., Ledoux, T., Léger, M. and Coupaye, Thierry, 2008. FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. *annals of telecommunications - annales des télécommunications*, 64(1-2), pp.45–63.

DeRemer, F. and Kron, H., 1975. Programming-in-the large versus programming-in-the-small. *ACM SIGPLAN Notices*, 10(6), pp.114–121.

Duquesne, P. and Bryce, C., 2008. A language model for dynamic code updating. *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades - HotSWUp '08*, p.1.

Endler, M. and Wei, J., 1992. Programming generic dynamic reconfigurations for distributed applications. In *International Workshop on Configurable Distributed Systems*. pp. 68–79.

Escoffier, C., Hall, R.S. and Lalanda, P., 2007. iPOJO: an Extensible Service-Oriented Component Framework. In *IEEE International Conference on Services Computing (SCC 2007)*. IEEE, pp. 474–481.

Fabry, R., 1976. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press.

Favre, J.-M., 1997. Understanding-in-the-large. In *Proceedings Fifth International Workshop on Program Comprehension. IWPC'97*. IEEE Comput. Soc. Press, pp. 29–38.

Fielding, R.T., 2000. *Architectural styles and the design of network-based software architectures*. Doctoral dissertation, University of California, Irvine.

Futuyma, D., 2009. *Evolution, Second Edition*, Sinauer Associates Inc.

Gacek, C., Abd-allah, A., Clark, B. and Boehm, B., 1995. On the Definition of Software System Architecture.

Gama, K. and Donsez, D., 2008. Using the service coroner tool for diagnosing stale references in the OSGi platform. In *Proceedings of the ACM/IFIP/USENIX international middleware conference companion on Middleware '08 Companion - Companion '08*. New York, New York, USA: ACM Press, p. 58.

Garcia, I., Pedraza, G., Debbabi, B., Lalanda, P. and Hamon, C., 2010. Towards a Service Mediation Framework for Dynamic Applications. In *IEEE Asia-Pacific Services Computing Conference*. IEEE, pp. 3–10.

Garlan, D., 2002. *Software Architecture*, John Wiley & Sons, Inc.

Garlan, D., Monroe, R.T. and Wile, D., 2000. Acme: architectural description of component-based systems. In G. T. Leavens & M. Sitaraman, eds. *Foundations of component-based systems*. New York, NY, USA: Cambridge University Press, pp. 47–67.

Genesereth, M.R., 1983. An overview of meta-level architecture. In *Proceedings of the Third National Conference on Artificial Intelligence*. pp. 119–124.

Gomaa, H. and Hussein, M., 2004. Software reconfiguration patterns for dynamic evolution of software architectures. *Architecture*, 0, pp.79–88.

Gupta, D., Jalote, P. and Barua, G., 1996. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2), pp.120–131.

Hammer, M., 2009. *How To Touch a Running System: Reconfiguration of Stateful Components*. Doctoral dissertation, Faculty of Mathematics, Computer Science and Statistics, LMU München.

Hicks, M., 2001. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August 2001. Winner of the 2002 ACM SIGPLAN Doctoral Dissertation award.

Hicks, M. and Nettles, S., 2005. Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6), pp.1049–1096.

Hirsch, D., Inverardi, Paolo and Montanari, U., 1998. Graph grammars and constraint solving for software architecture styles. *Proceedings of the third international workshop on Software architecture - ISAW '98*, pp.69–72.

Hjálmtýsson, G. and Gray, R., 1998. Dynamic C ++ Classes A lightweight mechanism to update code in a running program. In *In proceedings of the USENIX 1998 annual technical conference*.

Hofmeister, C., Nord, R. and Soni, D., 1999. *Applied software architecture*, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Hohpe, G. and Woolf, B., 2004. *Enterprise integration patterns: designing, building, and deploying messaging solutions*, Addison-Wesley.

Huhns, M.N. and Singh, M.P., 2005. Service-oriented computing: key concepts and principles. *IEEE Internet Computing*, 9(1), pp.75–81.

ISO/IEEE, 2006. ISO/IEC 14764 IEEE Std 14764-2006 International Standard - ISO/IEC 14764 IEEE Std 14764-2006 Software Engineering - Software Life Cycle Processes - Maintenance.

Kon, F. and Campbell, R.H., 2000. Dependence Management in Component-Based Distributed Systems. *System*, (March), pp.1–11.

Kon, F., Costa, F., Blair, G. and Campbell, R.H., 2002. The case for reflective middleware. *Communications of the ACM*, 45(6).

Kramer, J. and Magee, J., 1990. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering*, 16(11), pp.1293–1306.

Kruchten, P., 1995. Architecture blueprints--the "4+1" view model of software architecture. In *Tutorial proceedings on TRI-Ada "91 Ada"s role in global markets: solutions for a changing complex world - TRI-Ada '95*. New York, New York, USA: ACM Press, pp. 540–555.

Lau, K. and Elizondo, P.V., 2005. Exogenous connectors for software components. *Component-Based Software*, pp.90–106.

Lehman, M M., Ramil, J.F., Wernick, P.D., Perry, D E. and Turski, W.M., 1997. Metrics and Laws of Software Evolution - The Nineties View. In *METRICS '97 Proceedings of the 4th International Symposium on Software Metrics*. Washington, DC, USA: IEEE Computer Society, p. 20.

Lehman, M.M., 1980. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9), pp.1060–1076.

Lehman, Meir and Ramil, J., 2001. Rules and Tools for Software Evolution Planning and Management. *Annals of Software Engineering*, 11(1), pp.15–44.

Luckham, D.C. and Vera, J., 1995. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9), pp.717–734.

Maes, P., 1987. Concepts and experiments in computational reflection. *ACM SIGPLAN Notices*, 22(12), pp.147–155.

Magee, J., Dulay, N., Eisenbach, S. and Kramer, J., 1995. Specifying distributed software architectures. *Annals of Physics*, 54(2), pp.137–153.

Magee, J., Kramer, J. and Sloman, M., 1989. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, 15(6), pp.663–675.

Magee, Jeff and Kramer, Jeff, 1996. Dynamic structure in software architectures. *ACM SIGSOFT Software Engineering Notes*, 21(6), pp.3–14.

Maier, M.W., Emery, D. and Hilliard, R., 2001. Software architecture: introducing IEEE Standard 1471. *Computer*, 34(4), pp.107–109.

Maier, Mark W., Emery, D. and Hilliard, R., 2004. ANSI/IEEE 1471 and systems engineering. *Systems Engineering*, 7(3), pp.257–270.

Malabarba, S., Pandey, R., Gragg, J., Barr, E. and Barnes, J.F., 2000. Runtime Support for Type-Safe Dynamic Java Classes. In E. Bertino, ed. *ECOOP 00 Proceedings of the 14th European Conference on ObjectOriented Programming*. Springer-Verlag, pp. 337–361.

Malenfant, J., Jacques, M. and Demers, F., 1996. A tutorial on behavioral reflection and its implementation. In *Proceedings of Reflection*. pp. 1–20.

Mamone, S., 1994. The IEEE standard for software maintenance. *ACM SIGSOFT Software Engineering Notes*, 19(1), pp.75–76.

Marino, J. and Rowley, M., 2010. *Understanding SCA (Service Component Architecture) (Google eBook)*, Pearson Education.

McKenney, P.E. and Slingwine, J.D., 1998. Read-Copy-Update: Using execution history to solve concurrency problems. In *Proc. 10th IASTED Int'l Conf. Parallel and Distributed Computing and Systems*.

Medvidovic, N. and Taylor, R.N., 2000. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1), pp.70–93.

Medvidovic, Nenad and Taylor, Richard N., 1997. A framework for classifying and comparing architecture description languages. *ACM SIGSOFT Software Engineering Notes*, 22(6), pp.60–76.

Mehta, N.R., Medvidovic, N. and Phadke, S., 2000. Towards a taxonomy of software connectors. In *Proceedings of the 2000 International Conference on Software Engineering ICSE*. ACM, pp. 178–187.

Mens, T., 2008. Introduction and Roadmap: History and Challenges of Software Evolution. In D. Serge & T. Mens, eds. *Software Evolution*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–11.

Meyer, B., 2003. The grand challenge of trusted components. In *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, pp. 660–667.

Mittermeir, R.T., 2002. Software evolution: let's sharpen the terminology before sharpening (out-of-scope) tools. In *Proceedings of the 4th international workshop on Principles of software evolution - IWPSE '01*. New York, New York, USA: ACM Press, p. 114.

Morrison, R. et al., 2007. A framework for supporting dynamic systems co-evolution. *Automated Software Engineering*, 14(3), pp.261–292.

Nierstrasz, O. and Dami, L., 1995. Component-Oriented Software Technology. In O. Nierstrasz & D. Tsichritzis, eds. *Object-Oriented Software Composition*. Prentice Hall, pp. 3–28.

Northrop, L. et al., 2006. Ultra-Large-Scale Systems - The Software Challenge of the Future W. Pollak, ed. *Technical report Software Engineering Institute Carnegie Mellon University ISBN*, 0.

Odersky, M. and Zenger, M., 2005. Scalable component abstractions. *ACM SIGPLAN Notices*, 40(10), p.41.

Van Ommering, R., Van der Linden, F., Kramer, Jeff and Magee, J., 2000. The Koala component model for consumer electronics software. *Computer*, 33(3), pp.78–85.

Oquendo, F. et al., 2004. ArchWare: Architecting Evolvable Software F. Oquendo, B. C. Warboys, & R. Morrison, eds. *Software Architecture*, 3047, pp.257–271.

Oreizy, P., Medvidovic, N. and Taylor, R.N., 1998. Architecture-based runtime software evolution. *Proceedings of the 20th International Conference on Software Engineering*, 1998, pp.177–186.

Papazoglou, M.P., 2003. Service-oriented computing: concepts, characteristics and directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering, 2003, WISE 2003*. IEEE Computer Society, pp. 3–12.

Perry, D. and Wolf, A., 1992. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), pp.40–52.

Perry, Dewayne E., 2008. Issues in Architecture Evolution: Using Design Intent in Maintenance and Controlling Dynamic Evolution. In *ECSA '08 Proceedings of the 2nd European conference on Software Architecture*. Berlin, Heidelberg: Springer Berlin Heidelberg, p. 1.

Pfister, C. and Szyperski, C., 1998. Why objects are not enough. In *CUC96: Component Based Software Engineering*. p. 141.

Pissias, P. and Coulson, G., 2008. Framework for quiescence management in support of reconfigurable multi-threaded component-based systems. *IET Software*, 2(4), p.348.

Premkumar, G. and Potter, M., 1995. Adoption of computer aided software engineering (CASE) technology. *ACM SIGMIS Database*, 26(2-3), pp.105–124.

Reiser, M., 1991. The Oberon system: user guide and programmer's manual.

Rinard, M., Cadar, C., Dumitran, D., Roy, D.M., Leu, T. and Beebee Jr., W.S., 2004. Enhancing server availability and security through failure-oblivious computing., p.21.

Royce, W.W., 1970. Managing the development of large software systems. In *Electronics*. Los Angeles, pp. 1–9.

Schwaber, K. and Beedle, M., 2001. *Agile Software Development with Scrum*, Prentice Hall.

Segal, M.E. and Frieder, O., 1988. Dynamic program updating in a distributed computer system. In *Proceedings. Conference on Software Maintenance, 1988*. IEEE Comput. Soc. Press, pp. 198–203.

Shaw, M. and Clements, P., 2006. The golden age of software architecture. *IEEE Software*, 23(2), pp.31–39.

Shaw, Mary, 1993. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *ICSE '93 Selected papers from the Workshop on Studies of Software Design*. London, UK: Springer-Verlag, pp. 17–32.

Shaw, Mary and Garlan, D., 1996. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall.

Smith, B.C., 1982. *Procedural reflection in programming languages*. PhD Thesis, Massachusetts Institute of Technology.

Sykes, D., Heaven, W., Magee, Jeff and Kramer, Jeff, 2008. From goals to components. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems - SEAMS '08*. New York, New York, USA: ACM Press, p. 1.

Szyperski, C., 1997. *Component Software: Beyond Object-Oriented Programming* First Edit., Addison-Wesley Pub (Sd).

Szyperski, C., 1992. Import is Not Inheritance Why We Need Both : Modules and Classes. In *ECOOP '92 Proceedings of the European Conference on Object-Oriented Programming*. Rennes, France: Springer-Verlag.

Szyperski, C., 2000. Modules and Components - Rivals or Partners. In L. Boszormenyi, Jurg Gutknecht, & G. Pomberger, eds. *The school of Niklaus Wirth: the art of simplicity*. Morgan Kaufmann, pp. 121–132.

Szyperski, C., Gruntz, D. and Murer, S., 2002. *Component software: beyond object-oriented programming* 2nd ed., Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Taylor, R. N., Medvidovic, N. and Dashofy, E.M., 2009. *Software Architecture: Foundations, Theory, and Practice*, Wiley.

Taylor, Richard N. and Van der Hoek, A., 2007. *Software Design and Architecture The once and future focus of software engineering*, IEEE.

Taylor, Richard N., Medvidovic, Nenad and Oreizy, Peyman, 2009. Architectural styles for runtime software adaptation. *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, pp.171–180.

Tisato, F., Savigni, A., Cazzola, W. and Sosio, A., 2001. Architectural Reflection Realising: Software Architectures via Reflective Activities. In *EDO '00 Revised Papers from the Second International Workshop on Engineering Distributed Objects*. Springer-Verlag London, UK, pp. 102–115.

Vandewoude, Y and Berbers, Y, 2004. Fresco: Flexible and Reliable Evolution System for Components. *Electronic Notes in Theoretical Computer Science*, 127(3), pp.197–205.

Vandewoude, Yves and Berbers, Yolande, 2005. DeepCompare: Static Analysis for Runtime Software Evolution. In CW Reports, volume CW405, Department of Computer Science, K.U.Leuven, Leuven, Belgium.

Vandewoude, Yves, Ebraert, P., Berbers, Yolande and D'Hondt, T., 2006. An alternative to Quiescence: Tranquility. In *2006 22nd IEEE International Conference on Software Maintenance*. IEEE, pp. 73–82.

Vandewoude, Yves, Ebraert, P., Berbers, Yolande and D'Hondt, T., 2007. Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. *IEEE Transactions on Software Engineering*, 33(12), pp.856–868.

VanGurp, J. and Bosch, J., 2002. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2), pp.105–119.

Virding, R., Wikström, C. and Williams, M., 1996. *Concurrent programming in ERLANG* 2nd ed. Joe Armstrong, ed., Hertfordshire, UK: Prentice Hall International (UK) Ltd.

Wang, Q., Shen, J., Wang, X. and Mei, H., 2006. A component-based approach to online software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(3), pp.181–205.

Wasserman, A., 1990. Tutorial on Software Design Techniques.

Wermelinger, M.A., 1999. *Specification of software architecture reconfiguration*. PhD Thesis, Faculdade de Ciencias e Tecnologia Departamento de Informatica, Universidade Nova de Lisboa.

Wirth, N., 1995. A plea for lean software. *Computer*, 28(2), pp.64–68.

Wirth, N., 1977. Modula: A language for modular multiprogramming. *Software: Practice and Experience*, 7(1), pp.1–35.

Wirth, N., 1992. *Project Oberon: The Design of an Operating System and Compiler*, Addison-Wesley Pub (Sd).

Wirth, N., 1988. The programming language oberon. *Software: Practice and Experience*, 18(7), pp.671–690.

Wirth, N., 1971. The programming language pascal. *Acta Informatica*, 1(1), pp.35–63.